PATENT

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Application of: Sachin Mullick, et al.	Technology Center: 2100
Serial No.: 10/668.467 Confirm: 2942	Group Art Unit: 2161
Filed : 09/23/2003	Examiner: Bibbee, Jared M
For: Multi-Threaded Write Interface and Methods for Increasing the Single File Read and Write Throughput of a File Server	Atty. Dkt. No.: 10830.0100.NPUS00

APPEAL BRIEF TO THE BOARD OF PATENT APPEALS AND INTERFERENCES

Commissioner for Patents
PO Box 1450
Alexandria, Virginia 22313-1450

Sir:

This Appeal Brief is in support of Appellants' Notice of Appeal filed April 13, 2010, from the final Official Action dated Jan. 13, 2010. Please apply the Appeal Brief fee of $540 paid on Oct. 22, 2008 for the first appeal in this case to the present Appeal Brief. (See 1204.01 Reinstatement of Appeal [R-3].) Please deduct any deficiency in any required fee from EMC Corporation Deposit Account No. 05-0889.

I. REAL PARTY IN INTEREST

The real party in interest is EMC Corporation, by virtue of assignments recorded at Reel 014554 Frame 0097 and 014880 Frame 0711.

II. RELATED APPEALS AND INTERFERENCES

There are no related appeals or interferences.

III. STATUS OF THE CLAIMS

Claims 1-73 have been presented for examination.

Claims 29-31 and 55-57 have been canceled.

Claims 1-28, 32-54 and 58-73 have been finally rejected.

Claims 1-28, 32-54, 58-65, 67-71, and 73 are being appealed.

IV. STATUS OF AMENDMENTS

A Reply to Final Official Action was filed on April 13, 2010. This reply did not request any amendment to the specification or claims. An Advisory Action dated April 22, 2010 indicated that this request for reconsideration did not place the application in condition for allowance.

V. SUMMARY OF CLAIMED SUBJECT MATTER

The invention of appellants' independent claim 1 is a method of operating a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8) for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to a file (FIG. 14; page 29, line 22 to page 30, line 3). (Appellants' specification, page 3, lines 19-21.) The method includes the network file server responding to a concurrent write request from a client by obtaining a lock for the file (step 101 in FIG. 11), and then preallocating a metadata block for the file (step 102 in FIG. 11), and then releasing the lock for the file (step 103 in FIG. 11); and then asynchronously writing to the file (step 104 in FIG. 11); and then obtaining the lock for the file (step 106 in FIG. 11); and then committing the metadata block to the file (step 107 in FIG. 11); and then releasing the lock for the file (step 108 in FIG. 11). (Appellants' specification, page 3 line 21 to page 4 line 2; page 27 lines 3-23.) Appellants' FIG. 11 is reproduced below.

The invention of appellants' independent claim 13 is a method of operating a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8) for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to a file (FIG. 14; page 29, line 22 to page 30, line 3). (Appellants' specification, page 4, lines 3-4.) The method includes the network file server responding to a concurrent write request from a client by preallocating a block for the file (step 102 in FIG. 11); and then asynchronously

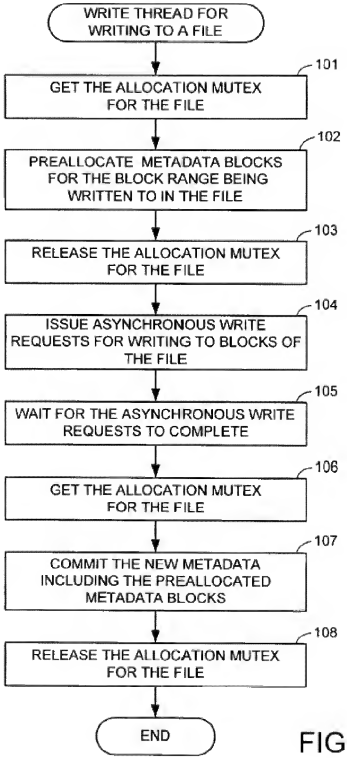

WRITE THREAD FOR WRITING TO A FILE
101
GET THE ALLOCATION MUTEX FOR THE FILE
102
PREALLOCATE METADATA BLOCKS FOR THE BLOCK RANGE BEING WRITTEN TO IN THE FILE
103
RELEASE THE ALLOCATION MUTEX FOR THE FILE
104
ISSUE ASYNCHRONOUS WRITE REQUESTS FOR WRITING TO BLOCKS OF THE FILE
105
WAIT FOR THE ASYNCHRONOUS WRITE REQUESTS TO COMPLETE
106
GET THE ALLOCATION MUTEX FOR THE FILE
107
COMMIT THE NEW METADATA INCLUDING THE PREALLOCATED METADATA BLOCKS
108
RELEASE THE ALLOCATION MUTEX FOR THE FILE
END

FIG. 11

writing to the file (step 104 in FIG. 11); and then committing the block to the file (step 107 in FIG. 11). (Appellants' specification, page 4, lines 4-7; page 27 lines 4-12 and 14-20.) The asynchronous writing to the file includes a partial write to a new block (123 in FIG. 13) that has been copied at least in part from an original block (121 in FIG. 13) of the file. (Appellants' specification, page 4, lines 7-9; page 28 line 22 to page 29 line 7.) The method includes checking a partial block conflict queue (73 in FIG. 4; page 14 lines 19-22; page 15 lines 20-22) for a conflict with a concurrent write to the new block (step 151 in FIG. 17; page 34 lines 7-10), and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block (step 156 in FIG. 17; page 34 line 24 to page 35 line 2), and then performing the partial write to the new block (step 157 in FIG. 17; page 35 lines 2-5). (Appellants' specification, page 4, lines 7-13.) Appellants' FIGS. 4, 13 and 17 are reproduced below.

The invention of appellants' independent 15 is a method of operating a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8) for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to a file (FIG. 14; page 29, line 22 to page 30, line 3). (Appellants' specification, page 4, lines 14-15.) The method includes the network file server responding to a concurrent write request from a client by preallocating a metadata block for the file (step 102 in FIG. 11), and then asynchronously writing to the file (step 104 in FIG. 11), and then committing the metadata

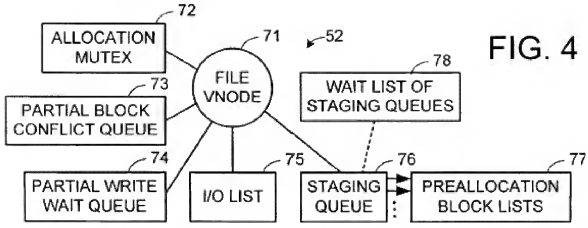

72
ALLOCATION MUTEX
71
52
FILE VNODE
78
WAIT LIST OF STAGING QUEUES
73
PARTIAL BLOCK CONFLICT QUEUE
74
PARTIAL WRITE WAIT QUEUE
75
I/O LIST
76
STAGING QUEUE
77
PREALLOCATION BLOCK LISTS

FIG. 4

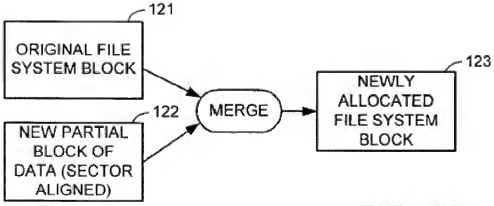

121
ORIGINAL FILE SYSTEM BLOCK
122
NEW PARTIAL BLOCK OF DATA (SECTOR ALIGNED)
MERGE
123
NEWLY ALLOCATED FILE SYSTEM BLOCK

FIG. 13

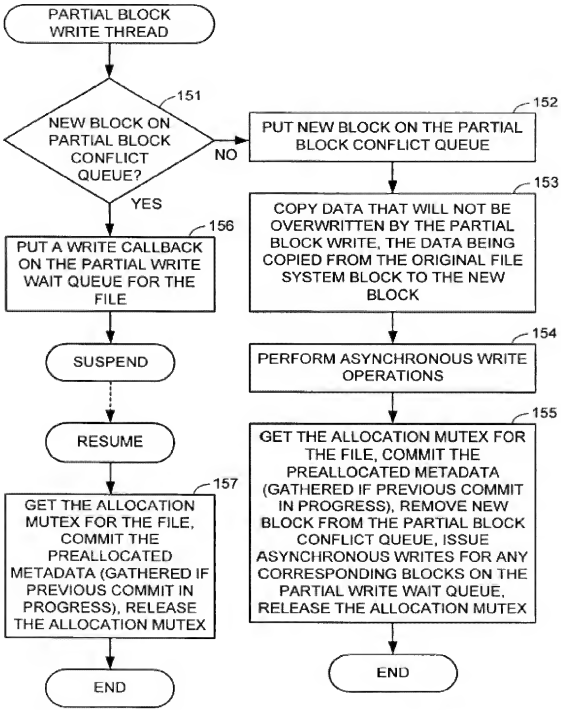

PARTIAL BLOCK WRITE THREAD
151
NEW BLOCK ON PARTIAL BLOCK CONFLICT QUEUE?
NO
YES
152
PUT NEW BLOCK ON THE PARTIAL BLOCK CONFLICT QUEUE
153
COPY DATA THAT WILL NOT BE OVERWRITTEN BY THE PARTIAL BLOCK WRITE, THE DATA BEING COPIED FROM THE ORIGINAL FILE SYSTEM BLOCK TO THE NEW BLOCK
154
PERFORM ASYNCHRONOUS WRITE OPERATIONS
155
GET THE ALLOCATION MUTEX FOR THE FILE, COMMIT THE PREALLOCATED METADATA (GATHERED IF PREVIOUS COMMIT IN PROGRESS), REMOVE NEW BLOCK FROM THE PARTIAL BLOCK CONFLICT QUEUE, ISSUE ASYNCHRONOUS WRITES FOR ANY CORRESPONDING BLOCKS ON THE PARTIAL WRITE WAIT QUEUE, RELEASE THE ALLOCATION MUTEX
END
156
PUT A WRITE CALLBACK ON THE PARTIAL WRITE WAIT QUEUE FOR THE FILE
SUSPEND
RESUME
157
GET THE ALLOCATION MUTEX FOR THE FILE, COMMIT THE PREALLOCATED METADATA (GATHERED IF PREVIOUS COMMIT IN PROGRESS), RELEASE THE ALLOCATION MUTEX
END

FIG. 17

block to the file (step 107 in FIG. 11). (Appellants' specification, page 4, lines 15-18; page 27 lines 4-12 and 14-20.) The method further includes gathering together preallocated metadata blocks for a plurality of client write requests to the file (step 117 in FIG. 12), and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file (step 106 in FIG. 11), committing the gathered preallocated metadata blocks for the plurality of client write requests to the file (step 107 in FIG. 11; step 118 in FIG. 12), and then releasing the lock for the file (step 108 in FIG. 11). (Appellants' specification, page 4, lines 18-23; page 27 lines 14-23; page 28, lines 9-21.) Appellants' FIG. 12 is reproduced below.

The invention of appellants' independent 25 is a method of operating a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8) for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent read and write access (page 15 lines 17-19) to a file (FIG. 14; page 29, line 22 to page 30, line 3). The method includes the network file server responding to a concurrent write request from a client by preallocating a metadata block for the file (step 102 in FIG. 11), and then asynchronously writing to the file (step 104 in FIG. 11), and then committing the metadata block to the file (step 107 in FIG. 11). (Appellants' specification, page 27 lines 4-12 and 14-20.) The network file server includes disk storage (29 in FIGS. 1 and 2; page 11 lines 3-6; page 12 lines 19-21) containing a file system (54 in FIG. 2 and FIG. 3; page 12 lines 6-9 and 19-21; page 13 lines 11-16), and a file system cache (51 in FIG. 2 and FIG. 3; page 12 lines 21-23; page 13 lines 8-16) storing data of blocks (134,

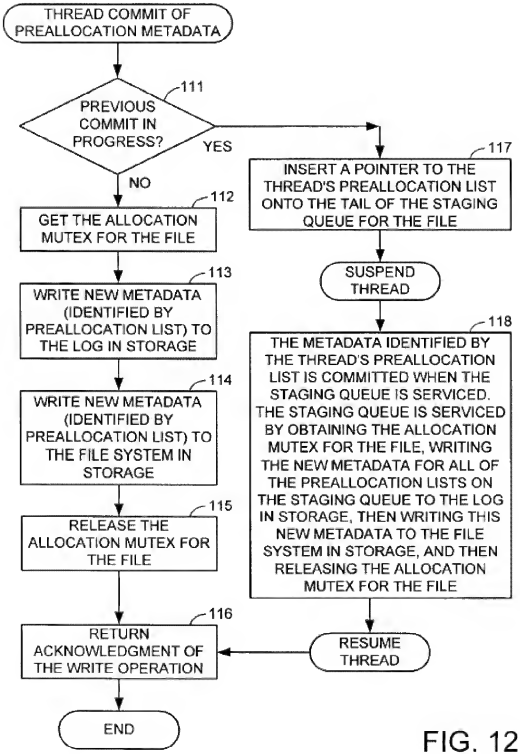

THREAD COMMIT OF PREALLOCATION METADATA
111
PREVIOUS COMMIT IN PROGRESS?
YES
NO
117
INSERT A POINTER TO THE THREAD'S PREALLOCATION LIST ONTO THE TAIL OF THE STAGING QUEUE FOR THE FILE
SUSPEND THREAD
112
GET THE ALLOCATION MUTEX FOR THE FILE
113
WRITE NEW METADATA (IDENTIFIED BY PREALLOCATION LIST) TO THE LOG IN STORAGE
114
WRITE NEW METADATA (IDENTIFIED BY PREALLOCATION LIST) TO THE FILE SYSTEM IN STORAGE
115
RELEASE THE ALLOCATION MUTEX FOR THE FILE
118
THE METADATA IDENTIFIED BY THE THREAD'S PREALLOCATION LIST IS COMMITTED WHEN THE STAGING QUEUE IS SERVICED. THE STAGING QUEUE IS SERVICED BY OBTAINING THE ALLOCATION MUTEX FOR THE FILE, WRITING THE NEW METADATA FOR ALL OF THE PREALLOCATION LISTS ON THE STAGING QUEUE TO THE LOG IN STORAGE, THEN WRITING THIS NEW METADATA TO THE FILE SYSTEM IN STORAGE, AND THEN RELEASING THE ALLOCATION MUTEX FOR THE FILE
116
RETURN ACKNOWLEDGMENT OF THE WRITE OPERATION
RESUME THREAD
END

FIG. 12

135, 138, 139 in FIG. 14; page 29, line 22 to page 30, line 3) of the file. The method includes the network file server responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache. (Steps 515 and 516 in FIG. 10; page 23 lines 19-23 and page 24 lines 7-15.) The method further includes the network file server responding to read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale. (Steps 92, 97, 513, 98 in FIG. 10; page 23 lines 5-17 page 24 lines 16-22.) Appellants' FIG. 10 is reproduced below.

The invention of appellants' independent claim 32 is a method of operating a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8) for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to a file (FIG. 14; page 29, line 22 to page 30, line 3). (Appellants' specification, page 5, lines 1-2.) The method includes the network file server responding to a concurrent write request from a client by executing a write thread (FIG. 11). (Appellants' specification, page 4, lines 3-4.). Execution of the write thread includes obtaining an allocation mutex for the file (step 101 in FIG. 11), and then preallocating new metadata blocks that need to be allocated for writing to the file (step 102 in FIG. 11), and then releasing the allocation mutex for the file (step 103 in FIG.

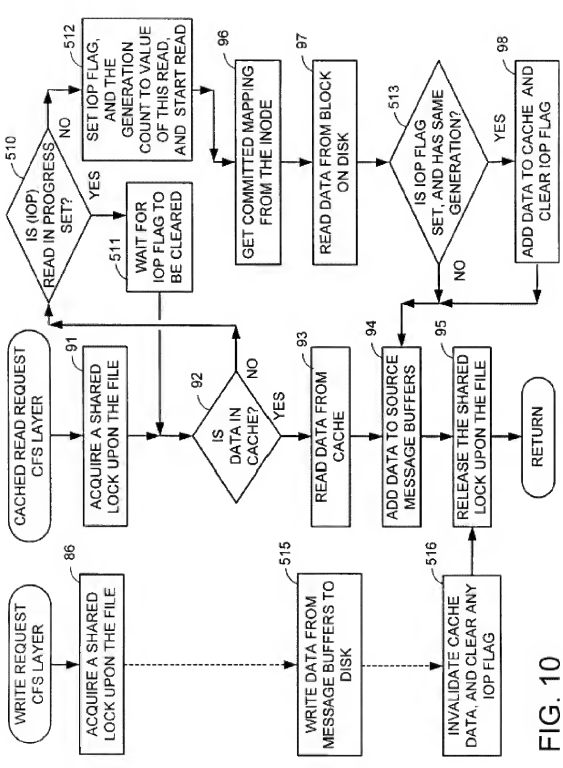

WRITE REQUEST CFS LAYER
86
ACQUIRE A SHARED LOCK UPON THE FILE
CACHED READ REQUEST CFS LAYER
91
ACQUIRE A SHARED LOCK UPON THE FILE
92
IS DATA IN CACHE?
NO
YES
93
READ DATA FROM CACHE
94
ADD DATA TO SOURCE MESSAGE BUFFERS
95
RELEASE THE SHARED LOCK UPON THE FILE
RETURN
510
IS (IOP) READ IN PROGRESS SET?
YES
NO
511
WAIT FOR IOP FLAG TO BE CLEARED
512
SET IOP FLAG, AND THE GENERATION COUNT TO VALUE OF THIS READ, AND START READ
96
GET COMMITTED MAPPING FROM THE INODE
97
READ DATA FROM BLOCK ON DISK
513
IS IOP FLAG SET, AND HAS SAME GENERATION?
NO
YES
98
ADD DATA TO CACHE AND CLEAR IOP FLAG
515
WRITE DATA FROM MESSAGE BUFFERS TO DISK
516
INVALIDATE CACHE DATA, AND CLEAR ANY IOP FLAG

FIG. 10

11); and then issuing asynchronous write requests for writing to the file (step 104 in FIG. 11), waiting for callbacks indicating completion of the asynchronous write requests (step 105 in FIG. 11), and then obtaining the allocation mutex for the file (step 106 in FIG. 11), and then committing the preallocated metadata blocks (step 107 in FIG. 11), and then releasing the allocation mutex for the file (step 108 in FIG. 11). (Appellants' specification, page 4, lines 4-10; page 27 lines 3-23.)

The invention of appellants' independent claim 33 is a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8). (Appellants' specification, page 5, line 11.) The network file server includes storage (29 in FIGS. 1 and 2; page 11 lines 3-6; page 12 lines 19-21) for storing a file (FIG. 14; page 29, line 22 to page 30, line 3), and at least one processor (26, 27, 28 in FIG. 1; page 11 lines 4-6) coupled to the storage for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to the file. (Appellants' specification, page 5, lines 12-13.) The network file server is programmed for responding to a concurrent write request from a client by obtaining a lock for the file (step 101 in FIG. 11), and then preallocating a metadata block for the file (step 102 in FIG. 11), and then releasing the lock for the file (step 103 in FIG. 11), and then asynchronously writing to the file (step 104 in FIG. 11), and then obtaining the lock for the file (step 106 in FIG. 11), and then committing the metadata block to the file (step 107 in FIG. 11), and then releasing the lock for the file (step 108 in FIG. 11). (Appellants' specification, page 5, lines 13-18; page 27 lines 3-23.)

The invention of appellants' independent claim 47 is a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8). (Appellants' specification, page 5, line 19.) The network file server includes storage (29 in FIGS. 1 and 2; page 11 lines 3-6; page 12 lines 19-21) for storing a file (FIG. 14; page 29, line 22 to page 30, line 3), and at least one processor (26, 27, 28 in FIG. 1; page 11 lines 4-6) coupled to the storage for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to the file. (Appellants' specification, page 5, lines 20-21.) The network file server is programmed for responding to a concurrent write request from a client by preallocating a block for the file (step 102 in FIG. 11); and then asynchronously writing to the file (step 104 in FIG. 11); and then committing the block to the file (step 107 in FIG. 11). (Appellants' specification, page 5, line 21 to page 6, line 1; page 27 lines 4-12 and 14-20.) The network file server includes a partial block conflict queue (73 in FIG. 4; page 14 lines 19-22; page 15 lines 20-22) for indicating a concurrent write to a new block that is being copied at least in part from an original block of the file. (Appellants' specification, page 6, lines 1-3.) The network file server is programmed for responding to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict (step 151 in FIG. 17; page 34 lines 7-10), and upon finding an indication of a conflict, waiting until resolution of the conflict with the concurrent write to the new block of the file (step 156 in FIG. 17; page 34 line 24 to page 35 line 2), and then performing the partial write to the new block of the file (step 157 in FIG. 17; page 35 lines 2-5). (Appellants' specification, page 6, lines 3-7.)

The invention of appellants' independent claim 49 is a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8). (Appellants' specification, page 6, line 8.) The network file server includes storage (29 in FIGS. 1 and 2; page 11 lines 3-6; page 12 lines 19-21) for storing a file (FIG. 14; page 29, line 22 to page 30, line 3), and at least one processor (26, 27, 28 in FIG. 1; page 11 lines 4-6) coupled to the storage for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to the file. (Appellants' specification, page 6, lines 9-10.) The network file server is programmed for responding to a concurrent write request from a client by preallocating a metadata block for the file (step 102 in FIG. 11), and then asynchronously writing to the file (step 104 in FIG. 11), and then committing the metadata block to the file (step 107 in FIG. 11). (Appellants' specification, page 6, lines 10-13; page 27 lines 4-12 and 14-20.) The network file server is programmed for gathering together preallocated metadata blocks for a plurality of client write requests to the file (step 117 in FIG. 12), and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file (step 106 in FIG. 11), committing the gathered preallocated metadata blocks for the plurality of client write requests to the file (step 107 in FIG. 11; step 118 in FIG. 12), and then releasing the lock for the file (step 108 in FIG. 11). (Appellants' specification, page 6, lines 13-18; page 27 lines 14-23; page 28, lines 9-21.)

The invention of appellants' independent 51 is a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8). The network file server includes disk

storage (29 in FIGS. 1 and 2; page 11 lines 3-6; page 12 lines 19-21) containing a file system (54 in FIG. 2 and FIG. 3), and a file system cache (51 in FIG. 2 and FIG. 3; page 12 lines 21-23; page 13 lines 8-16) storing data of blocks of a file (FIG. 14; page 29, line 22 to page 30, line 3) in the file system. (Appellants' specification, page 6, lines 20-22.) The network file server is programmed for responding to a concurrent write request from a client (23, 24, 25 in FIG. 1; page 11 lines 3-4) by preallocating a metadata block for the file (step 102 in FIG. 11); and then asynchronously writing to the file (step 104 in FIG. 11), and then committing the metadata block to the file (step 107 in FIG. 11). (Appellants' specification, page 27 lines 4-12 and 14-20.) The network file server is further programmed for responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache. (Steps 515 and 516 in FIG. 10; page 23 lines 19-23 and page 24 lines 7-15.) The network file server is programmed for responding to concurrent read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale. (Steps 92, 97, 513, 98 in FIG. 10; page 23 lines 5-17 page 24 lines 16-22.)

The invention of appellants' independent claim 58 is a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8). (Appellants' specification, page

6, lines 19-20.) The network file server includes storage (29 in FIGS. 1 and 2; page 11 lines 3-6; page 12 lines 19-21) for storing a file (FIG. 14; page 29, line 22 to page 30, line 3), and at least one processor (26, 27, 28 in FIG. 1; page 11 lines 4-6) coupled to the storage for providing clients (23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to the file. (Appellants' specification, page 6, lines 20-22.) The network file server is programmed with a write thread (FIG. 11) for responding to a concurrent write request from a client by obtaining an allocation mutex for the file (step 101 in FIG. 11), and then preallocating new metadata blocks that need to be allocated for writing to the file (step 102 in FIG. 11), and then releasing the allocation mutex for the file (step 103 in FIG. 11), and then issuing asynchronous write requests for writing to the file (step 101 in FIG. 11), waiting for callbacks indicating completion of the asynchronous write requests (step 101 in FIG. 11), and then obtaining the allocation mutex for the file (step 106 in FIG. 11); and then committing the preallocated metadata blocks (step 107 in FIG. 11), and then releasing the allocation mutex for the file (step 108 in FIG. 11). (Appellants' specification, page 6, line 22 to page 7, line 6; page 27 lines 3-23.)

The invention of appellants' independent claim 61 is a network file server (21 in appellants' FIG. 1; appellants' specification, page 11, lines 4-8). (Appellants' specification, page 7, line 7.) The network file server includes storage (29 in FIGS. 1 and 2; page 11 lines 3-6; page 12 lines 19-21) for storing a file (FIG. 14; page 29, line 22 to page 30, line 3), and at least one processor (26, 27, 28 in FIG. 1; page 11 lines 4-6) coupled to the storage for providing clients

(23, 24, 25 in FIG. 1; page 11 lines 3-4) with concurrent write access (page 15 lines 17-19) to the file. (Appellants' specification, page 7, lines 8-9.) The network file server is programmed for responding to a concurrent write request from a client by preallocating a block for writing to the file (step 102 in FIG. 11), asynchronously writing to the file (step 104 in FIG. 11), and then committing the preallocated block (step 107 in FIG. 11). (Appellants' specification, page 7, lines 9-12; page 27 lines 4-12 and 14-20.) The network file server also includes an uncached write interface (63 in FIG. 3), a file system cache (51 in FIG. 2 and FIG. 3), and a cached read-write interface (61 in FIG. 3). (Appellants' specification, page 7, lines 12-13; page 13 line 8 to page 14 line 17.) The uncached write interface bypasses the file system cache for sector-aligned write operations (FIG. 3; step 85 in FIG. 5), and the network file server is programmed to invalidate cache blocks in the file system cache including sectors being written to by the uncached write interface (FIG. 6, steps 88 and 89). (Appellants' specification, page 7, lines 13-16; page 13 lines 17-22; page 17 lines 12-15; page 18 lines 8-14.) Appellants' FIGS. 3, 5, and 6 are reproduced below.

It is respectfully submitted that none of appellants' claims contain any "means plus function" or "step plus function" as permitted by 35 U.S.C. 112, sixth paragraph.

The invention of appellants' dependent claims 2 and 34 the file (FIG. 14) further includes a hierarchy of blocks including an inode block (131 in FIG. 14) of metadata, data blocks (132, 134, 135, 138, 139 in FIG. 14) of file data, and indirect blocks of metadata (133, 136, 137 in

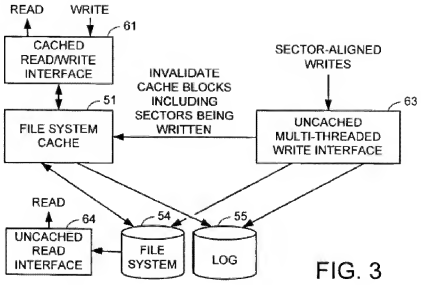

READ
WRITE
61
CACHED READ/WRITE INTERFACE
SECTOR-ALIGNED WRITES
INVALIDATE CACHE BLOCKS INCLUDING SECTORS BEING WRITTEN
51
FILE SYSTEM CACHE
63
UNCACHED MULTI-THREADED WRITE INTERFACE
READ
64
54
55
UNCACHED READ INTERFACE
FILE SYSTEM
LOG

FIG. 3

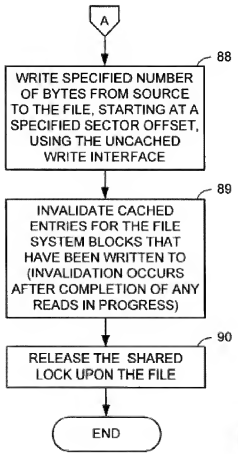

A
88
WRITE SPECIFIED NUMBER OF BYTES FROM SOURCE TO THE FILE, STARTING AT A SPECIFIED SECTOR OFFSET, USING THE UNCACHED WRITE INTERFACE
89
INVALIDATE CACHED ENTRIES FOR THE FILE SYSTEM BLOCKS THAT HAVE BEEN WRITTEN TO (INVALIDATION OCCURS AFTER COMPLETION OF ANY READS IN PROGRESS)
90
RELEASE THE SHARED LOCK UPON THE FILE
END

FIG. 6

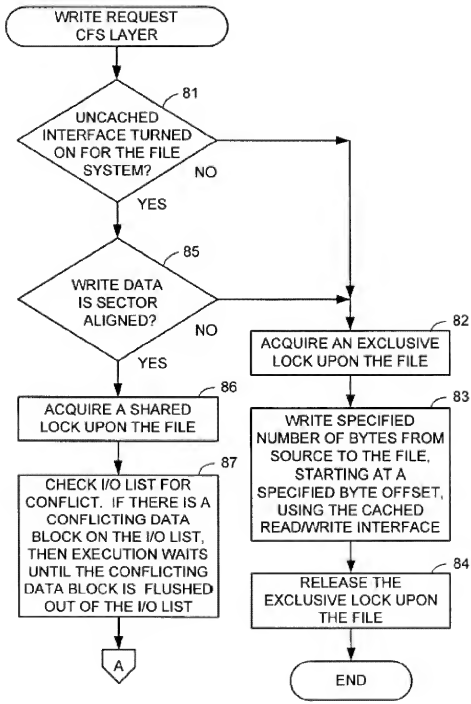

WRITE REQUEST CFS LAYER
81
UNCACHED INTERFACE TURNED ON FOR THE FILE SYSTEM?
NO
YES
85
WRITE DATA IS SECTOR ALIGNED?
NO
YES
82
ACQUIRE AN EXCLUSIVE LOCK UPON THE FILE
83
WRITE SPECIFIED NUMBER OF BYTES FROM SOURCE TO THE FILE, STARTING AT A SPECIFIED BYTE OFFSET, USING THE CACHED READ/WRITE INTERFACE
84
RELEASE THE EXCLUSIVE LOCK UPON THE FILE
END
86
ACQUIRE A SHARED LOCK UPON THE FILE
87
CHECK I/O LIST FOR CONFLICT. IF THERE IS A CONFLICTING DATA BLOCK ON THE I/O LIST, THEN EXECUTION WAITS UNTIL THE CONFLICTING DATA BLOCK IS FLUSHED OUT OF THE I/O LIST
A

FIG. 5

FIG. 14), and wherein the metadata block for the file is an indirect block of metadata. (Appellants' specification, page 14 lines 1-5, page 15 lines 6-8 and 17-18, page 24 lines 8-9, page 27 lines 4-6, page 29 line 22 to page 30 line 3).

The invention of appellants' dependent claims 3 and 35 further includes copying data from an original indirect block of the file (121 in FIG. 13; 136 in FIG. 14 and FIG. 15) to the metadata block for the file (123 in FIG. 13; 141 in FIG. 16), the original indirect block of the file having been shared between the file and a read-only version of the file. (Appellants' specification, page 15 lines 8-11; page 22 line 22 to page 31 line 7; page 28 line 22 to page 29 line 7.)

The inventions of appellants' dependent claims 4 and 36 further include concurrent writing for more than one client to the metadata block for the file. (Appellants' specification, page 15, lines 17-22.)

The inventions of appellants' dependent claims 5 and 37 further include asynchronous writing to the file including a partial write to a new block (123 in FIG. 13) that has been copied at least in part from an original block (121 in FIG. 13) of the file, and wherein the method further includes checking a partial block conflict queue (73 in FIG. 4; page 14 lines 19-22; page 15 lines 17-22) for a conflict with a concurrent write to the new block (step 151 in FIG. 17; page 34 lines 7-10), and upon failing to find an indication of a conflict with a concurrent write to the new

block, preallocating the new block (step 102 in FIG. 11), copying at least a portion of the original block of the file to the new block (step 153 in FIG. 17), and performing the partial write to the new block (step 154 in FIG. 17). (Appellants' specification, page 27 lines 4-6, page 28 line 22 to page 29 line 7, page 34 line 7 to page 35 line 5.)

The inventions of appellants' dependent claims 6 and 38 are similar to claim 13 to the extent that claims 6 and 38 further define that the asynchronous writing to the file includes a partial write to a new block (123 in FIG. 13) that has been copied at least in part from an original block (121 in FIG. 13) of the file, and wherein the method further includes checking a partial block conflict queue (73 in FIG. 4; page 14 lines 19-22; page 15 lines 17-22) for a conflict with a concurrent write to the new block (step 151 in FIG. 17; page 34 lines 7-10), and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block (step 156 in FIG. 17; page 34 line 24 to page 35 line 2), and then performing the partial write to the new block (step 157 in FIG. 17; page 35 lines 2-5).

The inventions of appellants' dependent claims 11 and 45 are similar to claim 15 to the extent that claims 11 and 45 further define gathering together preallocated metadata blocks for a plurality of client write requests to the file (step 117 in FIG. 12), and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining the lock for the file (step 106 in FIG. 11), committing the gathered preallocated metadata blocks for

the plurality of client write requests to the file (step 107 in FIG. 11; step 118 in FIG. 12), and then releasing the lock for the file (step 108 in FIG. 11). (Appellants' specification, page 4, lines 18-23; page 27 lines 14-23; page 28, lines 9-21.)

The inventions of appellants' dependent claims 12, 16, 46, and 50 further include checking whether a previous commit is in progress (step 111 in FIG. 12) after asynchronously writing to the file (steps 103 and 104 in FIG. 11) and before obtaining the lock for the file for committing the metadata block to the file (step 112 or step 118 in FIG. 12), and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on a staging queue (76 in FIG. 4) for the file (step 117 in FIG. 12). (Appellants' specification, page 28 lines 1-4 and 9-21.)

The inventions of appellants' dependent claim 17 is similar to claim 25 to the extent that claim 17 further defines the network file server responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache. (Steps 515 and 516 in FIG. 10; page 23 lines 19-23 and page 24 lines 7-15.)

The invention of appellants' dependent claim 18 is similar to claim 25 to the extent that claim 18 further defines the network file server responding to read requests for file blocks not

found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale. (Steps 92, 97, 513, 98 in FIG. 10; page 23 lines 5-17 page 24 lines 16-22.)

The inventions of appellant's dependent claims 19, 26 and 52 further include the network file server checking a read-in-progress flag for a file block (step 510 in FIG. 10) upon finding that the file block is not in the file system cache (step 92 in FIG. 10), and upon finding that the read-in-progress flag indicates that a prior read of the file block is in progress from the file system in the disk storage, waiting for completion of the prior read of the file block from the file system in the disk storage (step 511 in FIG. 10), and then again checking whether the file block is in the file system cache (step 92 in FIG. 10). (Appellants' specification, page 22 line 19 to page 23 line 9.)

The inventions of appellant's dependent claims 20, 27 and 53 further includes the network file server setting a read-in-progress flag for a file block (step 512 in FIG. 10) upon finding that the file block is not in the file system cache (step 92 in FIG. 10) and then beginning to read the file block from the file system in disk storage (step 512 in FIG. 10), clearing the read-in-progress flag upon writing to the file block on disk (steps 515 and 516 in FIG. 10), and inspecting the read-in-progress flag (step 513 in FIG. 10) to determine whether the file block has

become stale due a concurrent write to the file block. (Appellants' specification, page 23 line 10 to page 24 line 6.)

The inventions of appellant's dependent claim 21, 28 and 54 further include the network file server maintaining a generation count (step 512 in FIG. 10) for each read of a file block from the file system in the disk storage in response to a read request for a file block that is not in the file system cache (step 92 in FIG. 10), and checking whether a file block having been read from the file system in the disk storage has become stale by checking whether the generation count for the file block having been read from the file system is the same as the generation count for the last read request for the same file block (step 513 in FIG. 10).

The invention of appellants' dependent claim 59 is similar to claim 61 to the extent that claim 59 further defines an uncached write interface (63 in FIG. 3), a file system cache (51 in FIG. 2 and FIG. 3) and a cached read-write interface (61 in FIG. 3; appellants' specification, page 7, lines 12-13; page 13 line 8 to page 14 line 17) wherein the uncached write interface bypasses the file system cache for sector-aligned write operations. (FIG. 3; step 85 in FIG. 5; page 13 lines 17-23; page 17 lines 12-15.)

The invention of appellants' dependent claim 60 is similar to claim 61 to the extent that claim 60 defines that the network file server is further programmed to invalidate cache blocks in

the file system cache including sectors being written to by the uncached write interface. (FIG. 6, steps 88 and 89; appellants' specification, page 18 lines 8-14.)

VI. GROUNDS OF REJECTION TO BE REVIEWED ON APPEAL

1. Whether claims 1-4, 10, 11, 15, 22, 32, 33-36, 44, 45, 49, 58, and 61-65 and 67-71 and 73 are unpatentable under 35 U.S.C. 102(e) as being anticipated by Burns et al. U.S. Patent 6,925,515 B2.

2. Whether claims 5-9, 12-14, 16-21, 23-28, 37-43, 46-48, and 50-54 are unpatentable under 35 U.S.C. 103(a) over Burns et al. U.S. Patent 6,925,515 B2 in view of Marcotte U.S. Patent 6,449,614 B1.

3. Whether claims 59 and 60 are unpatentable under 35 U.S.C. 103(a) over Burns et al. U.S. Patent 6,925,515 B2 in view of Xu et al. US 6,324,581 B1.

VII. ARGUMENT

1. Claims 1-4, 10, 11, 15, 22, 32, 33-36, 44, 45, 49, 58, 61-65, 67-71, and 73 are patentable under 35 U.S.C. 102(e) and are not anticipated by Burns et al. U.S. Patent 6,925,515 B2.

"For a prior art reference to anticipate in terms of 35 U.S.C. § 102, every element of the claimed invention must be identically shown in a single reference." Diversitech Corp. v. Century Steps, Inc., 7 U.S.P.Q.2d 1315, 1317 (Fed. Cir. 1988), quoted in In re Bond, 910 F.2d 831, 15 U.S.P.Q.2d 1566, 1567 (Fed. Cir. 1990) (vacating and remanding Board holding of anticipation; the elements must be arranged in the reference as in the claim under review, although this is not an *ipsis verbis* test).

Claim Interpretation

With respect to appellants' claim 1, the word "then" appears at the end of each of steps (a), (b), (c), (d), (e), and (f). The word "then" cannot be rendered meaningless. Thus, the appellants construe the steps of claim 1 to be performed in a sequence in the order recited. In addition, step (a) is different from step (e), and step (c) is different from step (g).

"While not an absolute rule, all claim terms are presumed to have meaning in a claim." Innova/Pure Water v. Safari Water Filtration Sys., Inc., 381 F.3d 1111, 1119 (Fed. Cir. 2004)(defendant's claim construction impermissibly read the term "operatively" out of the

phrase "operatively connected"). Moreover, if the specification does not reveal any special definition for the phrase "; and then", then the phrase must be construed according to its ordinary meaning that the term would have to a person of ordinary skill in the art in question at the time of the invention. Phillips v. AWH Corp., 415 F.3d 1303, 1312-13 (Fed. Cir. 2005)(en banc). Dictionaries are among the many tools that can assist the court in determining the meaning of particular terminology to those of skill in the art of the invention. Id., at 1318.

Appellants respectfully submit that the ordinary meaning of the phrase "X, and then Y" is that Y is "following next after [X] in order". This ordinary meaning also avoids an issue of "double inclusion" that arises if step (a) were construed to be the same step as step (e), and step (c) were construed to be the same step as step (g).. Appellants respectfully submit that it is unreasonable to construe claim 1 so as to raise an issue of "double inclusion." See MPEP 2173.05(o) Double Inclusion. Appellants respectfully submit that similar language in their other claims should be construed in the same fashion.

Claims 1, 10, 33, 44, 62, 68

Appellants' claim 1 calls for a network file server responding to a concurrent write request from a client for access to a file by a sequence of seven specific steps performed in a specific order. The network file server responds by:

(a) obtaining a lock for the file and then

(b) preallocating a metadata block for the file; and then

(c) releasing the lock for the file; and then

(d) asynchronously writing to the file; and then

(e) obtaining the lock for the file; and then

(f) committing the metadata block to the file in the data storage; and then

(g) releasing the lock for the file.

The appellants' drawings show these steps (a) to (g) in FIG. 11 in boxes 101, 102, 103, 104-105, 106, 107, and 108, respectively, as described in appellants' specification on page 27 lines 3-23.

In appellants' view, Burns does not disclose a network file server computer responding to a concurrent write request from a client by performing the appellants' claimed sequence of seven steps. Instead, Burns discloses a network file server computer that responds to concurrent write requests by performing a different sequence of steps depending on the "locking mode" associated with the request. The citation on page 3 of the final Official Action to Burns col. 10 lines 14-30, however, indicates that the Official Action has identified the "locking mode" and the kind of write operation in Burns that is most pertinent to the subject matter of appellants' claim 1.

Burns (FIG. 3) discloses a Client/Server distributed file system on a storage area network (SAN). (Col. 5, lines 64-65.) The metadata is stored separately from the data. The metadata is stored in a file server cluster. The data is stored on shared disks of the SAN. (Col. 7, lines 1-9.) In a preferred embodiment, the clients of the distributed file system include a DBMS server, and a plurality of web servers. The DBMS server may update a web page, and the web servers may read the web page as the update occurs. The DBMS server obtains an exclusive "producer lock"

on the web page. The producer lock enables the holder to write data, and allocate and cache data for an "out of place" write that writes the data to a different physical storage location than from which it was read. By performing an "out-of-place" write, the old data still exists and is available to clients. Once the writer completes the write and releases the producer lock, the previous data is invalidated and the clients are informed of the new location of the data. Clients can then read the new data from storage when needed, and the server reclaims the old data blocks. (Burns, col. 5 lines 31-54; col. 10 lines 13-29.)

Page 3 of the final Official Action cites Burns col. 10 lines 14-30 for appellants' step (b) of preallocation, and also for appellants' step (c) of releasing the lock for the file. Page 3 of the final Official Action cites Burns col. 10 lines 30-38 for appellants' step (d) for asynchronously writing to the file. However, Burns col. 10 lines 14-30 do not disclose that the lock for the file (obtained in the first step (a)) is released after step (b) and before step (d), nor do Burns col. 10 lines 14-30 disclose that the lock for the file is obtained again in step (e) and released again in step (g), as recited in appellants' claim 1.

With respect to appellants' step (b) of "preallocating a metadata block for the file," page 3 lines 1-3 of the final Official Action says: "Note that the file is allocated before the write takes place making this a pre-allocation." However, the fact that the file is allocated before the write takes place is not a sufficient disclosure of preallocation of a metadata block in response to the client write request and after obtaining a lock for the file, as defined in claim 1 and many of appellants' other independent claims. For example, for the "in-place" write of Burns, the file and the storage of the file to receive the new data for the write operation is allocated before the client

request for the "in-place" write and before a shared lock is placed on the file in response to the client request for the "in-place" write.

With respect to appellants' step (c) of "releasing the lock for the file," page 3 lines 3-4 of the final Official Action says: "Note that on[c]e the file has been allocated it is unlocked for write." However, Burns col. 10, lines 14-30, does not disclose that the file server responds to a client write request by locking the file for allocation and then unlocking it for the write, as defined in appellants' claim 1. Instead, Burns col. 10 lines 14-30 disclose that "for database and parallel applications, the write privilege is granted on a shared lock and must be differentiated from allocation. Write, in this case, means an in-place write, where data are written back to the same physical location from which they were, or could have been, read. This is to be differentiated from out-of-place write, where portions of a file are written to physical storage locations that differ from the locations from which they were read."

More importantly, Burns col. 10, lines 14-30, do not disclose the network file server responding to a concurrent write request from a client by obtaining a lock for the file, and then preallocating a metadata block for the file, and then releasing the lock for the file, and then asynchronously writing to the file. Instead, Burns col. 10, lines 14-30 discloses and explicitly differentiates two different kinds of writes. The first kind of write is the "in-place" write, where the write privilege is granted on a shared lock, for example for database and parallel applications. The second kind of write is the out-of-place write, which should properly be considered an allocation followed by a write. As further disclosed in Burns col. 10 lines 35-37, allocation requires an exclusive lock.

Although Burns does not disclose the details of the two different kinds of writes, the "out-of-place" write of Burns could be performed by the following sequence in response to a first client write request:

a) obtaining the exclusive lock for the file; and then

b) obtaining allocations of data blocks for the new data; and then

c) writing new data to the allocated data blocks; and then

d) committing the allocated data blocks to the file in the data storage; and then

e) releasing the exclusive lock for the file.

The "in-place" write of Burns could be performed by the following sequence in response to a second client write request:

a) obtaining the shared lock on the file; and then

b) asynchronously writing data to the file; and then

c) releasing the shared lock on the file.

These hypothetical sequences for the "out-of-place" write and the "in-place" write of Burns are entirely consistent with the "Notes" in the final Official Action, yet the sequence of steps a) to g) of applicants' claim 1 is neither disclosed nor obvious from these hypothetical sequences. Thus, the appellants maintain that there is no disclosure or suggestion in Burns of

responding to a client write request by releasing the lock for the file after allocation of the data blocks (or preallocating a metadata block for the file) and before writing the new data to the file, nor is there any disclosure or suggestion of again obtaining the lock on the file after writing the new data to the file and before committing the allocated data blocks (or the metadata block) to the file in the data storage. In addition, Burns does not disclose details of how a block of data is allocated to the file, or committed to the file. For example, there is nothing in Burns disclosing that a metadata block is preallocated for the file, and then the file is asynchronously written to, and then the metadata block is committed to the file in data storage, as recited in appellants' claim 1.

Burns teaches away from releasing the lock for the file in step (c) and then asynchronously writing to the file in step (d) and then again obtaining the lock for the file in step (e), because Burns, as set out above, teaches that the file would be exclusively write-locked by the producer lock when a write operation adding a block to the file is performed on the file. See, for example, Burns col. 11, lines 47-49: "With C and P locks, the web servers are not updated until the P lock holder releases the lock, generally on closing the file." See also Burns, Abstract: "This system is implemented using two whole file locks: a producer lock P and a consumer lock C." (Emphasis added.)

Claims 2 and 34

With respect to appellants' dependent claims 2 and 34, page 3 of the Official Action cites Burns column 10 line 14 through column 11 line 14. Although Burns discloses that a P lock

holder can update location metadata at the server (col. 10 lines 49-53), Burns does not disclose details of this metadata structure such as an indirect block of metadata, as recited in appellants' claim 2.

Claims 3 and 35

With respect to appellants' dependent claims 3 and 35, page 3 of the Official Action cites Burns column 10 line 14 through column 11 line 14. However, Burns does not disclose details of this metadata structure such as an indirect block of metadata, or the use of this metadata structure such as copying data from an original indirect block of the file to the metadata block for the file, the original indirect block of the file having been shared between the file and a read-only version of the file, as recited in appellants' claim 3.

Claim 4 and 36

With respect to appellants' dependent claims 4 and 36, page 3 of the Official Action cites Burns, column 10 line 14 through column 11 line 14. However, Burns does not disclose concurrent writing for more than one client to a metadata block for the file. Instead, as discussed above, Burns discloses that a write operation that would change the allocation of a block requires an exclusive lock.

Claims 11 and 45

With respect to appellants' dependent claims 11 and 45, page 4 of the Official Action cites Burns column 10 line 14 through column 11 line 14. Burns, however, does not disclose gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining the lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file. Instead, if a client obtains a producer lock for a change in allocation of the file, this producer lock is exclusive and prevents other clients from obtaining a producer lock on the file. (See col. 10, lines 50-65.) The producer lock would be obtained and held before and while any new data blocks, and their metadata, would be preallocated and later committed. Therefore Burns teaches away from gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining the lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

Claims 15, 22, 49, 64, and 70

With respect to appellants' independent claims 15 and 49, as discussed above with reference to appellants' claim 11, Burns fails to disclose gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the

preallocated metadata blocks for the plurality of client write requests to the file by obtaining the lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

Claims 32, 58, 67, and 73

With respect to appellants' independent claims 32 and 58, see appellants' remarks above with reference to appellants' claim 1. In short, Burns does not disclose details of how a new metadata block is added to the file. There is no disclosure or suggestion in Burns of responding to a client write request by releasing the allocation mutex for the file after allocation of the data blocks (or preallocating a metadata block for the file) and before writing the new data to the file, nor is there any disclosure or suggestion of again obtaining the allocation mutex for the file after writing the new data to the file and before committing the allocated data blocks (or the metadata block) to the file in the data storage. Therefore, appellants respectfully submit that Burns does not disclose that new metadata bocks are added to the file by the appellants' recited sequence of the steps (a), (b), (c), (d) & (e), (f), (g), and (h).

Claim 61

With respect to appellants' independent claim 61, Burns discloses that a change in allocation requires an exclusive lock. Burns, col. 10, lines 35-37. In Burns, there can be multiple concurrent writers that write directly to the data storage of the SAN, thereby bypassing the server cluster. In this case, however, Database and Parallel Application Locking (instead of

File System Locking) is used to ensure data consistency. See the table at the top of column 10 in Burns. Thus, Burns does not disclose that the network file server includes both an uncached write interface and a cached write interface in which the network file server is further programmed to invalidate cache blocks in a file system cache including sectors being written to by an uncached write interface. Instead, Burns teaches a different way of ensuring consistency between client caches and the result of an "out-of-place" write that changes the allocation of the file. The "out-of-place" writer obtains an exclusive "producer lock" on the file, and then writes to the file. "The writer must release the P lock on close to publish the file. The server sends location updates to all clients that hold consumer locks 410. The clients immediately invalidate the affected blocks in their cache." (Burns, column 11, lines 34-38.) Thus, Burns teaches the network file server invalidating client caches when file system locking is used, rather than the network file server invalidating a file system cache of a cached read-write interface when an uncached write interface bypasses the file system cache by performing a sector-aligned write operation.

Claims 63 and 65

Appellants' dependent claims 63 and 65 are dependent upon claim 13, and therefore incorporate by reference the limitations of claim 13. Claim 13 is distinguished from Burns because, as recognized on page 8 of the final Official Action of January 13, 2010, Burns does not disclose that the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method includes

checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

Claims 65 and 71

Appellants' dependent claims 65 and 71 are dependent upon claim 25, and therefore incorporate by reference the limitations of claim 25. Claim 25 is distinguished from Burns because, as recognized on page 18 of the final Official Action of January 13. 2010, Burns does not disclose the network file server computer responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache, and Burns also does not disclose the network file server computer responding to read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale.

2. Claims 5-9, 12-14, 16-21, 23-28, 37-43, 46-48, and 50-54 are patentable under 35 U.S.C. 103(a) over Burns et al. U.S. Patent 6,925,515 B2 in view of Marcotte U.S. Patent 6,449,614 B1.

The policy of the Patent and Trademark Office has been to follow in each and every case the standard of patentability enunciated by the Supreme Court in Graham v. John Deere Co., 148 U.S.P.Q. 459 (1966). M.P.E.P. § 2141. As stated by the Supreme Court:

> Under § 103, the scope and content of the prior art are to be determined; differences between the prior art and the claims at issue are to be ascertained; and the level of ordinary skill in the pertinent art resolved. Against this background, the obviousness or nonobviousness of the subject matter is determined. Such secondary considerations as commercial success, long felt but unsolved needs, failure of others, etc., might be utilized to give light to the circumstances surrounding the origin of the subject matter sought to be patented. As indicia of obviousness or nonobviousness, these inquiries may have relevancy.

148 U.S.P.Q. at 467.

The problem that the inventor is trying to solve must be considered in determining whether or not the invention would have been obvious. The invention as a whole embraces the structure, properties and problems it solves. In re Wright, 848 F.2d 1216, 1219, 6 U.S.P.Q.2d 1959, 1961 (Fed. Cir. 1988).

Burns discloses a distributed file system supporting three kinds of locking systems. A first locking system is designed for sequential consistency with write-back caching, typical of distributed file systems. A second locking system is provided for sequential consistency with no caching for applications that manage their own caches. A third locking system implements a weaker consistency model with write-back caching, designed for efficient replication and distribution of data. Locks for replication are suitable for serving dynamic data on the Internet and other highly-concurrent applications. The selection of the appropriate lock protocol for each file is set using the file metadata. Further, a novel locking system is provided for the lock system implementing a weak consistency model with write back caching. This system is implemented utilizing two whole file locks: a producer lock P and a consumer lock C. Any client can hold a consumer lock and when holding a consumer lock can read data and cache data for read. The producer lock is only held by a single writer and a writer holding a producer lock can write data, allocate and cache data for writing. When a writer performs a write, the write is performed as an out-of-place write. An out-of-place write writes the data to a different physical storage location than from which it was read. By performing an out-of-place write the old data still exists and is available to clients. Once the writer completes the write and releases the producer lock the previous data is invalidated and the clients are informed of the new location of the data. Clients can then read the new data from storage when needed and the server reclaims the old data blocks. (See Burns, Abstract.)

Marcotte discloses an interface system and methods for asynchronously updating a share resource with locking facility. Tasks make updates requested by calling tasks to a shared resource serially in a first come first served manner, atomically, but not necessarily synchronously, such that a current task holding an exclusive lock on the shared resource makes the updates on behalf of one or more calling tasks queued on the lock. Updates waiting in a queue on the lock to the shared resource may be made while the lock is held, and others deferred for post processing after the lock is released. Some update requests may also, at the calling application's option, be executed synchronously. Provision is made for nested asynchronous locking. Data structures (wait_elements) describing update requests may queued in a wait queue for update requests awaiting execution by a current task, other than the calling task, currently holding an exclusive lock on the shared resource. Other queues are provided for queuing data structures removed from the wait queue but not yet processed; data structures for requests to unlock or downgrade a lock; data structures for requests which have been processed and need to be returned to free storage; and data structures for requests that need to be awakened or that describe post processing routines that are to be run while the lock is not held. (Marcotte, Abstract.)

As discussed above, various elements of the respective base claims are missing from Burns. Marcotte does not disclose these elements missing from Burns, so that the appellants' claimed invention does not result from the proposed combination of Burns and Marcotte.

Moreover, the appellants' claims define a substantial improvement over Burns and Marcotte by providing a new way of handling a write request that changes the allocation of the data blocks to a file, and this new way of handling such a write request solves a long-felt need for reducing contention during multi-threaded or multi-processor access to a shared file system.

Appellants respectfully submit that Burns does not address the problem of providing concurrent writes for the case of a write operation that changes the allocation of a block of data. Instead, Burns col. 10 lines 35-37 says: "The only guarantee multiple concurrent writers need is that the data does not change location, hence allocation requires an exclusive lock." Thus, appellants rely on their previously submitted evidence of a long-felt but unsolved problem that in a shared or parallel file system, the potential speed at which an application may execute is impaired by the need for file locking.

Chang et al. US 2005/0039049, cited by the Examiner, is evidence of a long-felt but unsolved problem that in a shared or parallel file system, the potential speed at which an application may execute is impaired by the need for file locking. Chang teaches that for "an application having its own serialization or locking mechanisms" (Chang, paragraph [0014]), "multiple processes may write to the same block of data within the file at approximately the same time as long as they are not changing the allocation of the block of data, i.e. either allocating the block, deallocating the block of data, or changing the block of data." (Chang, paragraph [0015].) The appellants' invention further solves this problem by enabling multiple

processes to write to the file at approximately the same time when updating the metadata structure associated with the file. The appellants cited additional evidence (on an IDS form filed on April 15, 2009) showing that shared or parallel file systems and their associated problems have been known for at least a decade prior to the filing of the appellants' patent application. See Row et al. U.S. Patent 5,163,131 filed Sept. 8, 1989 entitled Parallel I/O Network File Server Architecture, and Hitz et al. U.S. Patent 6,065,037, also having a priority date of Sep. 8, 1989. Copies of Chang, Row, and Hitz are included in appellants' Evidence Appendix IX.

With respect to appellants' dependent claims 5-9, 12, 16-21, 23-24, 37-43, and 50-54, these claims depend from the independent claims 1, 15, 33, and 49. Burns has been distinguished above with respect to the independent claims 1, 15, 33, and 49, and Marcotte does not provide the limitations of these independent claims that are missing from Burns. Therefore the dependent claims 5-9, 12, 16-21, 23-24, 37-43, and 50-54 are patentable over the proposed combination of Burns and Marcotte. It would not have been obvious for one of ordinary skill to combine Burns and Marcotte in the fashion as proposed in the Official Action and then modify that combination by adding the missing limitations.

Claims 5 and 37

Appellants' dependent claim 5 adds to claim 1 the limitations of "wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method further includes checking

a partial block conflict queue for a conflict with a concurrent write to the new block, and upon failing to find an indication of a conflict with a concurrent write to the new block, preallocating the new block, copying at least a portion of the original block of the file to the new block, and performing the partial write to the new block." Appellants' partial block conflict queue 73 is shown in appellants' FIG. 4 and described in appellant's specification on page 15 lines 17-22 as follows:

> The preallocation method allows concurrent writes to indirect blocks within the same file. Multiple writers can write to the same indirect block tree concurrently without improper replication of the indirect blocks. Two different indirect blocks will not be allocated for replicating the same indirect block. The write threads use the partial block conflict queue 73 and the partial write wait queue 74 to avoid conflict during partial block write operations, as further described below with reference to FIG. 13.

See also appellants' FIG. 13 and specification page 28 line 22 to page 29 line 7; and FIG. 17 and page 34 line 7 to page 35 line 5.

With respect to appellants' dependent claim 5, pages 8-9 of the final Official Action recognizes that Burns fails to explicitly recite the limitations added by dependent claim 5, and says that Marcotte teaches these limitations, citing column 13, lines 35 through col. 14, line 43. However, Marcotte column 13, lines 35 through col. 14, line 43 deals with managing an I/O device holding queue above a device for queuing pending I/O's if the number of I/O's issued to the device exceeds a threshold that is adjustable by an application. It is not seen where Marcotte discloses a partial write to a new block that has been copied at least in part from an original

block of the file. Nor is it seen where Marcotte discloses checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon failing to find an indication of a conflict with a concurrent write to the new block, preallocating the new block, copying at least a portion of the original block of the file to the new block, and performing the partial write to the new block.

"[R]ejections on obviousness grounds cannot be sustained by mere conclusory statements; instead, there must be some articulated reasoning with some rational underpinning to support the legal conclusion of obviousness." In re Kahn, 441 F. 3d 977, 988 (Fed. Cir. 2006). A fact finder should be aware of the distortion caused by hindsight bias and must be cautious of arguments reliant upon ex post reasoning. See KSR International Co. v. Teleflex Inc., 550 U.S. __, 82 USPQ2d 1385 (2007)), citing Graham, 383 U. S. at 36 (warning against a "temptation to read into the prior art the teachings of the invention in issue" and instructing courts to "guard against slipping into the use of hindsight.").

Claims 6 and 38

Appellants' claim 6 is similar to claim 5 in that it also adds to claim 1 the limitations of wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method further includes checking a partial block conflict queue for a conflict with a concurrent write to the new block. With respect to dependent claim 6, pages 8-9 of the final Official Action also recognizes that Burns fails to explicitly recite the limitations added by dependent claim 6, and says that Marcotte

teaches these limitations, again citing Marcotte column 13, lines 35 through col. 14, line 43. However, Marcotte column 13, lines 35 through col. 14, line 43 deals with managing an I/O device holding queue above a device for queuing pending I/O's if the number of I/O's issued to the device exceeds a threshold that is adjustable by an application. It is not seen where Marcotte discloses a partial write to a new block that has been copied at least in part from an original block of the file. Nor is it seen where Marcotte discloses checking a partial block conflict queue for a conflict with a concurrent write to the new block.

Claims 12 and 46

With respect to appellants' dependent claim 12, Burns does not further teach checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for the file for committing the metadata block to the file. As discussed above with reference to appellants' claim 1, in response to a concurrent write request from a client, Burns does not again obtain the lock for the file after the asynchronously writing to the file. Nor does Marcotte column 12, line 7 through column 13, line 32 disclose these limitations missing from Burns or specifically deal with committing "metadata blocks" to a file.

Claims 13, 14, 47, and 48

Appellants' independent claim 13 recites "wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method includes checking a partial block conflict queue for a conflict

with a concurrent write to the new block, and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block." Thus, Appellants' independent claim 13 is patentable over Burns in combination with Marcotte for the reasons given above with reference to appellants' claim 5. Marcotte does not discloses a partial write to a new block that has been copied at least in part from an original block of the file and checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon failing to find an indication of a conflict with a concurrent write to the new block, preallocating the new block, copying at least a portion of the original block of the file to the new block, and performing the partial write to the new block.

Claims 16 and 50

Appellants' dependent claim 16 adds to claim 15 the express limitations of claim 12 and therefore is patentable over Burns in combination with Marcotte for the reasons given above with reference to appellants' claims 12 and 15. Burns does not further teach checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for the file for committing the metadata block to the file.

Claim 17

Appellants' dependent claim 17 adds to claim 15 the limitations of "wherein the network file server includes disk storage containing a file system, and a file system cache storing data of

blocks of the file, and the method further includes the network file server computer responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache." Pages 13-14 of the Official Action recognize that Burns fails to explicitly recite the recited operations of "the network file server computer responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache." Page 14 of the Official Action cites Marcotte column 12 line 7 through column 13, line 32. However, the appellants' specific claim limitations are not disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte below with reference to claim 25.

Claim 18

Appellants' dependent claim 18 adds to claim 17 the limitations of "the network file server computer responding to read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale." Page 14 of the Official Action again cites Marcotte column 12 line 7 through column 13, line 32. However, the specific limitations of appellants' claim 18 are not

disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte below with reference to claim 25.

Claim 19

Appellants' dependent claim 19 adds to claim 18 the limitations of "the network file server computer checking a read-in-progress flag for a file block upon finding that the file block is not in the file system cache, and upon finding that the read-in-progress flag indicates that a prior read of the file block is in progress from the file system in the disk storage, waiting for completion of the prior read of the file block from the file system in the disk storage, and then again checking whether the file block is in the file system cache." Page 14-15 of the Official Action again cites Marcotte column 12 line 7 through column 13, line 32. However, the use of a read-in-progress flag as specifically recited in appellants' claim 19 is not disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte below with reference to claim 25.

Claim 20

Appellants' dependent claim 20 adds to claim 18 the limitations of "the network file server computer setting a read-in-progress flag for a file block upon finding that the file block is not in the file system cache and then beginning to read the file block from the file system in disk storage, clearing the read-in-progress flag upon writing to the file block on disk, and inspecting the read-in-progress flag to determine whether the file block has become stale due a concurrent

write to the file block." Page 15 of the Official Action again cites Marcotte column 12 line 7 through column 13, line 32. However, use of a read-in-progress flag as specifically recited in appellants' claim 20 is not disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte below with reference to claim 25.

Claim 21

Appellants' dependent claim 21 adds to claim 18 the limitations of "the network file server computer maintaining a generation count for each read of a file block from the file system in the disk storage in response to a read request for a file block that is not in the file system cache, and checking whether a file block having been read from the file system in the disk storage has become stale by checking whether the generation count for the file block having been read from the file system is the same as the generation count for the last read request for the same file block." Page 15 of the Official Action again cites Marcotte column 12 line 7 through column 13, line 32. However, the maintenance of a generation count as specifically recited in claim 21 is not disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte below with reference to claim 25.

Claims 25 and 51

With reference to appellants' independent claim 25, page 17 of the Official Action recognizes that Burns fails to explicitly recite the network file server responding to concurrent write requests by writing new data for specified blocks of the file to disk storage without writing

the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache. (See, e.g., appellants' FIG. 10, steps 515 and 516; appellants' spec., page 23 lines 19-23 and page 24 lines 7-15.) Page 17 of the Official Action further recognizes that Burns fails to explicitly recite the network file server responding to read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become state, and not writing to the file system cache a file block that has become stale. (See, e.g., appellants' FIG. 10, steps 92, 97, 513, 98; appellants' spec., page 23 lines 5-17; page 24 lines 16-22.) Page 18 of the Official Action cites Marcotte col. 12 line 7 through column 12, line 32 for all of these limitations not explicitly recited in Burns. However, it is not understood how all of the appellants' specific claim limitations are disclosed in Marcotte column 12 line 7 through column 13, line 32.

Marcotte column 12 line 7 through column 13, line 32, deals generally with maintaining a list of lock waiters. As shown in Marcotte FIG. 9, when a task waits for a lock, it queues its WAIT_ELEMENT to the lock using lock or wait routine 175 and also adds it WAIT_ELEMENT to a global list or queue 230 before it waits, and removes it from the global list 230 after it waits. As shown in Marcotte FIG. 11, do_wait 240 is executed each time a thread needs to go into a wait for a lock. Step 241 executes a lock_UpdateResource procedure. Step 242 waits on ecb; and upon receiving it, step 243 executes the lock_Update Resource procedure. Marcotte says that in this way, the task waiting on a lock 100 will only actually suspend itself on the WAIT

call. Adding and removing tasks (WAIT ELEMENTS) from global list 230 is done in a manner guaranteed to make the calling task wait. FIG. 12 shows that an add_to_global routine 245 (with waitp-arg) includes step 246 which determines if prevent flag is null; if so, step 248 posts an error code in the ecb field 108 of the WAIT_ELEMENT being processed; and, if not, step 247 adds the WAIT_ELEMENT (in this case, task 232) to the head of global list 230. FIG. 13 shows a remove_from_global routine 250 (with waitp=arg) including step 251 which determines if prevent flag 124 is null. If so, return code (rc) is set to zero; and if not, this WAIT_ELEMENT (say, 233) is removed from global list 230. In step 254, the return code (rc) is returned to the caller. FIG. 14 shows a return_wait routine 260 (with waitp=prc) including step 261 which determines if waitp is null. If not, the WAIT_ELEMENT pointed to by waitp is returned to free storage. Return wait 260 is the post processing routine for remove_from_global 250, and remove_from_global communicates the address of the WAIT_ELEMENT via its return code, that is input to return_wait (automatically by the resource update facility) as prc. Return_wait 260 returns the WAIT_ELEMENT to free storage. Since routine 260 is a post processing routine, the free storage return is NOT performed while holding the lock. This shows the benefits of a post processing routine, and passing the return value from a resource update routine to the post processing routine. FIG. 15 shows quiesce_global 265 wakes up all waiters and in step 267 tells them that the program is terminating due to error by way of prevent flag 124 being set to 1 in step 266. In step 268 pointer 231 and 234 are cleared so global list 230 is empty.

Thus, Marcotte's general teaching of a way of maintaining a list of lock waiters fails to disclose the appellants' specific limitations of the network file server responding to concurrent

write requests by writing new data for specified blocks of the file to disk storage without writing the new data for the specified blocks of the file to the file system cache, invalidating the specified blocks of the file in the file system cache, and responding to read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become state, and not writing to the file system cache a file block that has become stale.

"[R]ejections on obviousness grounds cannot be sustained by mere conclusory statements; instead, there must be some articulated reasoning with some rational underpinning to support the legal conclusion of obviousness." In re Kahn, 441 F. 3d 977, 988 (Fed. Cir. 2006). A fact finder should be aware of the distortion caused by hindsight bias and must be cautious of arguments reliant upon ex post reasoning. See KSR International Co. v. Teleflex Inc., 550 U.S. __, 82 USPQ2d 1385 (2007)), citing Graham, 383 U. S. at 36 (warning against a "temptation to read into the prior art the teachings of the invention in issue" and instructing courts to "guard against slipping into the use of hindsight.")

Claims 26 and 52.

Appellants' dependent claim 26 adds to claim 25 the limitations of "the network file server computer checking a read-in-progress flag for a file block upon finding that the file block is not in the file system cache, and upon finding that the read-in-progress flag indicates that a prior read of the file block is in progress from the file system in the disk storage, waiting for

completion of the prior read of the file block from the file system in the disk storage, and then again checking whether the file block is in the file system cache." Page 18 of the Official Action again cites Marcotte column 12 line 7 through column 13, line 32. However, the use a read-in-progress flag as specifically recited in appellants' claim 26 is not disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte above below with reference to claim 25.

Claims 27 and 53

Appellants' dependent claim 27 adds to claim 25 the limitations of "the network file server computer setting a read-in-progress flag for a file block upon finding that the file block is not in the file system cache and then beginning to read the file block from the file system in disk storage, clearing the read-in-progress flag upon writing to the file block on disk, and inspecting the read-in-progress flag to determine whether the file block has become stale due a concurrent write to the file block." Page 19 of the Official Action again cites Marcotte column 12 line 7 through column 13, line 32. However, the use of a read-in-progress flag as specifically recited in appellants' claim 27 is not disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte below with reference to claim 25.

Claims 28 and 54

Appellants' dependent claim 28 adds to claim 25 the limitations of "the network file server computer maintaining a generation count for each read of a file block from the file system in the disk storage in response to a read request for a file block that is not in the file system cache, and checking whether a file block having been read from the file system in the disk storage has become stale by checking whether the generation count for the file block having been read from the file system is the same as the generation count for the last read request for the same file block." Page 19 of the Official Action again cites Marcotte column 12 line 7 through column 13, line 32. However, the maintenance of a generation count as specifically recited in appellants' claim 28 is not disclosed in Marcotte column 12 line 7 through column 13, line 32. See the discussion of Marcotte below with reference to claim 25.

3. Claims 59 and 60 are patentable under 35 U.S.C. 103(a) over Burns et al. U.S. Patent 6,925,515 B2 in view of Xu et al. US 6,324,581 B1.

Appellants' dependent claims 59 and 60 are patentable over the proposed combination of Burns and Xu due to the limitations of their independent base claim 58. Burns is distinguished from the base claim 58 as discussed above with reference to claim 1. Xu is distinguished from the base claim 58 in a similar fashion Xu fails to disclose appellants' step c) of releasing the allocation mutex of the file [prior to the step d) of issuing asynchronous write requests for writing to the file], and Xu fails to disclose appellants' step f) of obtaining the allocation mutex for the file [after the step d) of issuing asynchronous write requests for writing to the file].

In view of the above, the rejection of the claims should be reversed.

Respectfully submitted,

/ Richard C. Auchterlonie /

Richard C. Auchterlonie
Reg. No. 30,607
NOVAK DRUCE & QUIGG, LLP
1000 Louisiana, 53rd Floor
Houston, TX 77002
713-571-3460 (Telephone)
713-456-2836 (Telefax)
Richard.Auchterlonie@novakdruce.com

VIII. CLAIMS APPENDIX

The claims involved in this appeal are as follows:

1. A method of operating a network file server computer for providing clients with concurrent write access to a file in data storage, the method comprising the network file server computer responding to a concurrent write request from a client by:

(a) obtaining a lock for the file; and then

(b) preallocating a metadata block for the file; and then

(c) releasing the lock for the file; and then

(d) asynchronously writing to the file; and then

(e) obtaining the lock for the file; and then

(f) committing the metadata block to the file in the data storage; and then

(g) releasing the lock for the file.

2. The method as claimed in claim 1, wherein the file further includes a hierarchy of blocks including an inode block of metadata, data blocks of file data, and indirect blocks of metadata, and wherein the metadata block for the file is an indirect block of metadata.

3. The method as claimed in claim 2, which further includes copying data from an original indirect block of the file to the metadata block for the file, the original indirect block of the file having been shared between the file and a read-only version of the file.

4. The method as claimed in claim 1, which further includes concurrent writing for more than one client to the metadata block for the file.

5. The method as claimed in claim 1, wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method further includes checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon failing to find an indication of a conflict with a concurrent write to the new block, preallocating the new block, copying at least a portion of the original block of the file to the new block, and performing the partial write to the new block.

6. The method as claimed in claim 1, wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method further includes checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

7. The method as claimed in claim 6, which further includes placing a request for the partial write in a partial write wait queue upon finding an indication of a conflict with a concurrent write to the new block, and performing the partial write upon servicing the partial write wait queue.

8. The method as claimed in claim 1, which further includes checking an input-output list for a conflicting prior concurrent access to the file, and upon finding a conflicting prior concurrent access to the file, suspending the asynchronous writing to the file until the conflicting prior concurrent access to the file is no longer conflicting.

9. The method as claimed in claim 8, which further includes providing a sector-level granularity of byte range locking for concurrent write access to the file by the suspending of the asynchronous writing to the file until the conflicting prior concurrent access is no longer conflicting.

10. The method as claimed in claim 1, which further includes writing the metadata block to a log in storage of the network file server computer for committing the metadata block for the file.

11. The method as claimed in claim 1, which further includes gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining the

lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

12. The method as claimed in claim 1, which further includes checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for the file for committing the metadata block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on a staging queue for the file.

13. A method of operating a network file server computer for providing clients with concurrent write access to a file in data storage, the method comprising the network file server computer responding to a concurrent write request from a client by:

(a) preallocating a block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the block to the file in the data storage;

wherein the asynchronous writing to the file includes a partial write to a new block that has been copied at least in part from an original block of the file, and wherein the method includes checking a partial block conflict queue for a conflict with a concurrent write to the new block, and upon finding an indication of a conflict with a concurrent write to the new block, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

14. The method as claimed in claim 13, wherein the method further includes placing a request for the partial write in a partial write wait queue upon finding an indication of a conflict with a concurrent write to the new block, and performing the partial write upon servicing the partial write wait queue.

15. A method of operating a network file server computer for providing clients with concurrent write access to a file, the method comprising the network file server computer responding to a concurrent write request from a client by:

(a) preallocating a metadata block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the metadata block to the file in the data storage;

wherein the method includes gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

16. The method as claimed in claim 15, which further includes checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for

the file for committing the block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on a staging queue for the file.

17. The method as claimed in claim 15, wherein the network file server computer includes disk storage containing a file system, and a file system cache storing data of blocks of the file, and the method further includes the network file server computer responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache.

18. The method as claimed in claim 17, which further includes the network file server computer responding to read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale.

19. The method as claimed in claim 18, which further includes the network file server computer checking a read-in-progress flag for a file block upon finding that the file block is not in the file system cache, and upon finding that the read-in-progress flag indicates that a prior read

of the file block is in progress from the file system in the disk storage, waiting for completion of the prior read of the file block from the file system in the disk storage, and then again checking whether the file block is in the file system cache.

20. The method as claimed in claim 18, which further includes the network file server computer setting a read-in-progress flag for a file block upon finding that the file block is not in the file system cache and then beginning to read the file block from the file system in disk storage, clearing the read-in-progress flag upon writing to the file block on disk, and inspecting the read-in-progress flag to determine whether the file block has become stale due to a concurrent write to the file block.

21. The method as claimed in claim 18, which further includes the network file server computer maintaining a generation count for each read of a file block from the file system in the disk storage in response to a read request for a file block that is not in the file system cache, and checking whether a file block having been read from the file system in the disk storage has become stale by checking whether the generation count for the file block having been read from the file system is the same as the generation count for the last read request for the same file block.

22. The method as claimed in claim 15, which further includes processing multiple concurrent read and write requests by pipelining the requests through a first processor and a

second processor, the first processor performing metadata management for the multiple concurrent read and write requests, and the second processor performing asynchronous reads and writes for the multiple concurrent read and write requests.

23. The method as claimed in claim 15, which further includes serializing the reads by delaying access for each read to a block that is being written to by a prior, in-progress write until completion of the write to the block that is being written to by the prior, in-progress write.

24. The method as claimed in claim 15, which further includes serializing the writes by delaying access for each write to a block that is being accessed by a prior, in-progress read or write until completion of the read or write to the block that is being accessed by the prior, in-progress read or write.

25. A method of operating a network file server computer for providing clients with concurrent read and write access to a file in data storage, the method comprising the network file server computer responding to a concurrent write request from a client by:

(a) preallocating a metadata block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the metadata block to the file in the data storage;

wherein the network file server computer includes disk storage containing a file system, and a file system cache storing data of blocks of the file, and the method includes the network

file server computer responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache, and

which includes the network file server computer responding to read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale.

26. The method as claimed in claim 25, which further includes the network file server computer checking a read-in-progress flag for a file block upon finding that the file block is not in the file system cache, and upon finding that the read-in-progress flag indicates that a prior read of the file block is in progress from the file system in the disk storage, waiting for completion of the prior read of the file block, and then again checking whether the file block is in the file system cache.

27. The method as claimed in claim 25, which further includes the network file server computer setting a read-in-progress flag for a file block upon finding that the file block is not in the file system cache and then beginning to read the file block from the file system in disk storage, clearing the read-in-progress flag upon writing to the file block on disk, and inspecting

the read-in-progress flag to determine whether the file block has become stale due to a concurrent write to the file block.

28. The method as claimed in claim 25, which further includes the network file server computer maintaining a generation count for each read of a file block from the file system in the disk storage in response to a read request for a file block that is not in the file system cache, and checking whether a file block having been read from the file system in the disk storage has become stale by checking whether the generation count for the file block having been read from the file system is the same as the generation count for the last read request for the same file block.

32. A method of operating a network file server computer for providing clients with concurrent write access to a file in data storage, the method comprising the network file server computer responding to a concurrent write request from a client by executing a write thread, execution of the write thread including:

(a) obtaining an allocation mutex for the file; and then

(b) preallocating new metadata blocks that need to be allocated for writing to the file; and then

(c) releasing the allocation mutex for the file; and then

(d) issuing asynchronous write requests for writing to the file;

(e) waiting for callbacks indicating completion of the asynchronous write requests; and then

(f) obtaining the allocation mutex for the file; and then

(g) committing the preallocated metadata blocks to the file in the data storage; and then

(h) releasing the allocation mutex for the file.

33. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) obtaining a lock for the file; and then

(b) preallocating a metadata block for the file; and then

(c) releasing the lock for the file; and then

(d) asynchronously writing to the file; and then

(e) obtaining the lock for the file; and then

(f) committing the metadata block to the file; and then

(g) releasing the lock for the file.

34. The network file server as claimed in claim 33, wherein the file further includes a hierarchy of blocks including an inode block of metadata, data blocks of file data, and indirect blocks of metadata, and wherein the metadata block for the file is an indirect block of metadata.

35. The network file server as claimed in claim 34, which is further programmed for copying data from an original indirect block of the file to the metadata block for the file, the original indirect block of the file having been shared between the file and a read-only version of the file.

36. The network file server as claimed in claim 33, which is further programmed for concurrent writing for more than one client to the metadata block for the file.

37. The network file server as claimed in claim 33, which further includes a partial block conflict queue for indicating a concurrent write to a new block that is being copied at least in part from an original block of the file, and wherein the network file server is further programmed to respond to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict, and upon failing to find an indication of a conflict, preallocating the new block, copying at least a portion of the original block of the file to the new block, and performing a partial write to the new block.

38. The network file server as claimed in claim 33, which further includes a partial block conflict queue for indicating a concurrent write to a new block that is being copied at least in part from an original block of the file, and wherein the network file server is further programmed to respond to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict, and upon finding an indication of a conflict, waiting until resolution of the conflict with the concurrent write to the new block, and then performing the partial write to the new block.

39. The network file server as claimed in claim 38, which further includes a partial write wait queue, and wherein the network file server is further programmed for placing a request for the partial write in the partial write wait queue upon finding an indication of a conflict, and performing the partial write upon servicing the partial write wait queue.

40. The network file server as claimed in claim 33, which is further programmed for maintaining an input-output list of concurrent reads and writes to the file, and when writing to the file, for checking the input-output list for a conflicting prior concurrent read or write access to the file, and upon finding a conflicting prior concurrent read or write access to the file, suspending the asynchronous writing to the file until the conflicting prior concurrent read or write access to the file is no longer conflicting.

41. The network file server as claimed in claim 40, which is further programmed so that the suspending of the asynchronous writing to the file until the conflicting prior concurrent read or write access to the file is no longer conflicting provides a sector-level granularity of byte range locking for concurrent write access to the file.

42. The network file server as claimed in claim 33, which is further programmed for maintaining an input-output list of concurrent reads and writes to the file, and when reading from the file, for checking the input-output list for a conflicting prior concurrent write access to the

file, and upon finding a conflicting prior concurrent write access to the file, suspending the reading to the file until the conflicting prior concurrent write access to the file is no longer conflicting.

43. The network file server as claimed in claim 42, which is further programmed so that the suspending of the reading to the file until the conflicting prior concurrent write access to the file is no longer conflicting provides a sector-level granularity of byte range locking for concurrent read access to the file.

44. The network file server as claimed in claim 33, which is further programmed for committing the metadata block for the file by writing the metadata block to a log in the storage.

45. The network file server as claimed in claim 33, which is further programmed for gathering together preallocated metadata blocks for a plurality of client requests for write access to the file, and committing together the preallocated metadata blocks for the plurality of client requests for access to the file by obtaining the lock for the file, committing the gathered preallocated metadata blocks for the plurality of client requests for write access to the file, and then releasing the lock for the file.

46. The network file server as claimed in claim 33, which further includes a staging queue for the file, and which is further programmed for checking whether a previous commit is in progress

after asynchronously writing to the file and before obtaining the lock for the file for committing the metadata block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on the staging queue for the file.

47. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) preallocating a block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the block to the file;

wherein the network file server includes a partial block conflict queue for indicating a concurrent write to a new block that is being copied at least in part from an original block of the file, and wherein the network file server is programmed for responding to a client request for a partial write to the new block by checking the partial block conflict queue for a conflict, and upon finding an indication of a conflict, waiting until resolution of the conflict with the concurrent write to the new block of the file, and then performing the partial write to the new block of the file.

48. The network file server as claimed in claim 47, which further includes a partial write wait queue, and wherein the network file server is programmed for placing a request for the partial

write in the partial write wait queue upon finding an indication of a conflict, and performing the partial write upon servicing the partial write wait queue.

49. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) preallocating a metadata block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the metadata block to the file;

wherein the network file server is programmed for gathering together preallocated metadata blocks for a plurality of client write requests to the file, and committing together the preallocated metadata blocks for the plurality of client write requests to the file by obtaining a lock for the file, committing the gathered preallocated metadata blocks for the plurality of client write requests to the file, and then releasing the lock for the file.

50. The network file server as claimed in claim 49, which is further programmed for checking whether a previous commit is in progress after asynchronously writing to the file and before obtaining the lock for the file for committing the metadata block to the file, and upon finding that a previous commit is in progress, placing a request for committing the metadata block to the file on a staging queue for the file.

51. A network file server comprising disk storage containing a file system, and a file system cache storing data of blocks of a file in the file system, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) preallocating a metadata block for the file; and then

(b) asynchronously writing to the file; and then

(c) committing the metadata block to the file;

wherein the network file server is further programmed for responding to concurrent write requests by writing new data for specified blocks of the file to the disk storage without writing the new data for the specified blocks of the file to the file system cache, and invalidating the specified blocks of the file in the file system cache, and

wherein the network file server is programmed for responding to concurrent read requests for file blocks not found in the file system cache by reading the file blocks from the file system in disk storage and then checking whether the file blocks have become stale due to concurrent writes to the file blocks, and writing to the file system cache a file block that has not become stale, and not writing to the file system cache a file block that has become stale.

52. The network file server as claimed in claim 51, which is further programmed for checking a read-in-progress flag for a file block upon finding that the file block is not in the file system cache, and upon finding that the read-in-progress flag indicates that a prior read of the file block is in progress from the file system in the disk storage, waiting for completion of the

prior read of the file block, and then again checking whether the file block is in the file system cache.

53. The network file server as claimed in claim 51, which is further programmed for setting a read-in-progress flag for a file block upon finding that the file block is not in the file system cache and then beginning to read the file block from the file system in disk storage, clearing the read-in-progress flag upon writing to the file block on disk, and inspecting the read-in-progress flag to determine whether the file block has become stale due to a concurrent write to the file block.

54. The network file server as claimed in claim 51, which is further programmed for maintaining a generation count for each read of a file block from the file system in the disk storage in response to a read request for a file block that is not in the file system cache, and checking whether a file block having been read from the file system in the disk storage has become stale by checking whether the generation count for the file block having been read from the file system is the same as the generation count for the last read request for the same file block.

58. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the

network file server is programmed with a write thread for responding to a concurrent write request from a client by:

(a) obtaining an allocation mutex for the file; and then

(b) preallocating new metadata blocks that need to be allocated for writing to the file; and then

(c) releasing the allocation mutex for the file; and then

(d) issuing asynchronous write requests for writing to the file;

(e) waiting for callbacks indicating completion of the asynchronous write requests; and then

(f) obtaining the allocation mutex for the file; and then

(g) committing the preallocated metadata blocks; and then

(h) releasing the allocation mutex for the file.

59. The network file server as claimed in claim 58, which further includes an uncached write interface, a file system cache and a cached read-write interface, and wherein the uncached write interface bypasses the file system cache for sector-aligned write operations.

60. The network file server as claimed in claim 59, wherein the network file server is further programmed to invalidate cache blocks in the file system cache including sectors being written to by the uncached write interface.

61. A network file server comprising storage for storing a file, and at least one processor coupled to the storage for providing clients with concurrent write access to the file, wherein the network file server is programmed for responding to a concurrent write request from a client by:

(a) preallocating a block for writing to the file;

(b) asynchronously writing to the file; and then

(c) committing the preallocated block;

wherein the network file server also includes an uncached write interface, a file system cache, and a cached read-write interface, wherein the uncached write interface bypasses the file system cache for sector-aligned write operations, and the network file server is programmed to invalidate cache blocks in the file system cache including sectors being written to by the uncached write interface.

62. The method as claimed in claim 1, which further includes a final step of returning to said client an acknowledgement of the writing to the file.

63. The method as claimed in claim 13, which further includes a final step of returning to said client an acknowledgement of the writing to the file.

64. The method as claimed in claim 15, which further includes a final step of returning to said client an acknowledgement of the writing to the file.

65. The method as claimed in claim 25, which further includes a final step of returning to said client an acknowledgement of the writing to the file.

67. The method as claimed in claim 32, which further includes a final step of returning to said client an acknowledgement of the writing to the file.

68. The method as claimed in claim 1, which further includes a final step of saving the file in disk storage of the network file server.

69. The method as claimed in claim 13, which further includes a final step of saving the file in disk storage of the network file server.

70. The method as claimed in claim 15, which further includes a final step of saving the file in disk storage of the network file server.

71. The method as claimed in claim 25, which further includes a final step of saving the file in the disk storage.

73. The method as claimed in claim 32, which further includes a final step of saving the file in disk storage of the network file server.

IX. EVIDENCE APPENDIX

Attached are copies of the following evidence:

1. Chang et al., U.S. Patent Application Publication US 2005/0039049 A1 published Feb. 17, 2005, cited by the Examiner as Ref. A on the Notice of References Cited PTO-892 attached to the Official Action dated Jan 13, 2009.

2. Row et al. U.S. Patent 5,163,131 issued Nov. 10, 1992, cited by the Appellants as Ref. 1 in the Information Disclosure Statement filed April 15, 2009.

3. Hitz et al. U.S. Patent 6,065,037 issued May 16, 2000, cited by the Appellants as Ref. 6 in the Information Disclosure Statement filed April 15, 2009.

US 20050039049A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2005/0039049 A1

Chang et al. (43) Pub. Date: Feb. 17, 2005

(54) METHOD AND APPARATUS FOR A MULTIPLE CONCURRENT WRITER FILE SYSTEM

(75) Inventors: Joon Chang, Austin, TX (US); Gerald Francis McBrearty, Austin, TX (US); Duyen M. Tong, Austin, TX (US)

Correspondence Address:
IBM CORP (YA)
C/O YEE & ASSOCIATES PC
P.O. BOX 802333
DALLAS, TX 75380 (US)

(73) Assignee: International Business Machines Corporation, Armonk, NY

(21) Appl. No.: 10/640,848

(22) Filed: Aug. 14, 2003

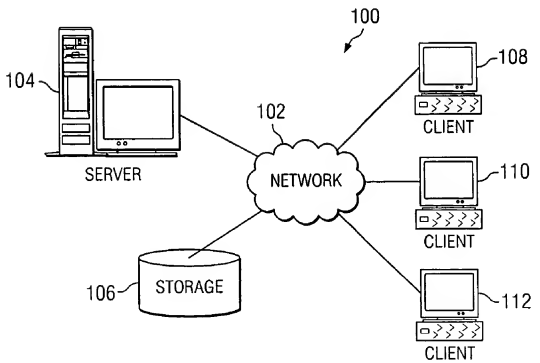
Publication Classification

(51) Int. Cl.[7] G06F 12/00

(52) U.S. Cl. 713/201

(57) ABSTRACT

A method and apparatus for a multiple concurrent writer file system are provided. With the method and apparatus, the metadata of a file includes a read lock, a write lock and a concurrent writer flag. If the concurrent writer flag is set, the file allows for multiple writers. That is, multiple processes may write to the same block of data within the file at approximately the same time as long as they are not changing the allocation of the block of data, i.e. either allocating the block, deallocating the block of data, or changing the size of the block of data. Multiple writers is facilitated by allowing processes performing write operations that do not require or result in a change to the allocation of data blocks in a file to use the read lock of a file rather than the write lock of the file. Software serialization or integrity mechanisms may be used to govern the manner by which these concurrent write operations have their results reflected in the file structure. Those processes performing write operations that do require or result in a change in the allocation of data blocks in a file must still acquire the write lock before performing their operation.



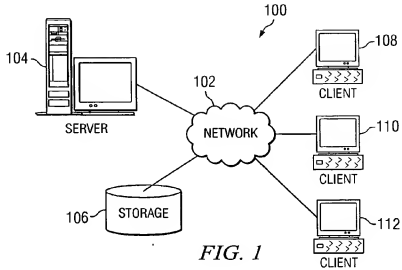

100
104
SERVER
102
NETWORK
108
CLIENT
110
CLIENT
112
CLIENT
106
STORAGE

FIG. 1

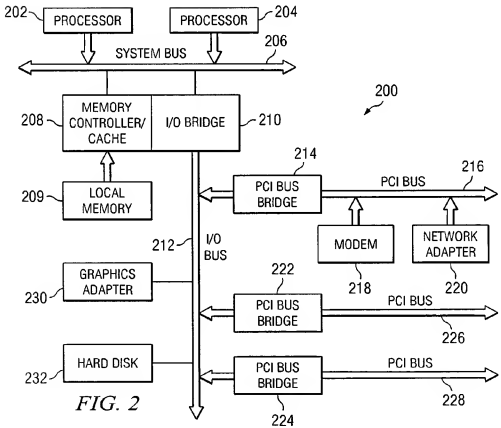

202
PROCESSOR
PROCESSOR
204
SYSTEM BUS
206
200
208
MEMORY CONTROLLER/ CACHE
I/O BRIDGE
210
214
216
209
LOCAL MEMORY
PCI BUS BRIDGE
PCI BUS
212
I/O BUS
MODEM
NETWORK ADAPTER
222
218
220
230
GRAPHICS ADAPTER
PCI BUS BRIDGE
PCI BUS
226
232
HARD DISK
PCI BUS BRIDGE
PCI BUS
228
224

FIG. 2

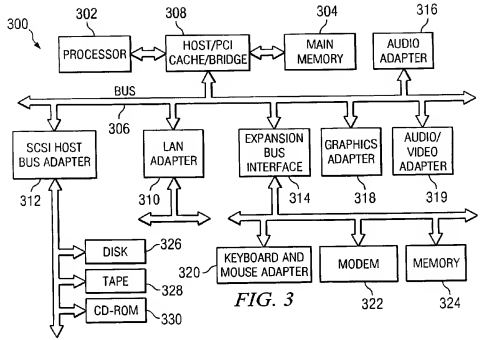

300
302
PROCESSOR
308
HOST/PCI CACHE/BRIDGE
304
MAIN MEMORY
316
AUDIO ADAPTER
BUS
306
SCSI HOST BUS ADAPTER
LAN ADAPTER
EXPANSION BUS INTERFACE
GRAPHICS ADAPTER
AUDIO/ VIDEO ADAPTER
312
310
314
318
319
DISK
326
TAPE
328
CD-ROM
330
320
KEYBOARD AND MOUSE ADAPTER
MODEM
MEMORY
322
324

FIG. 3

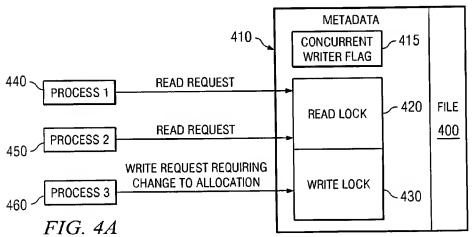

METADATA
410
CONCURRENT WRITER FLAG
415
440
PROCESS 1
READ REQUEST
READ LOCK
420
FILE
400
450
PROCESS 2
READ REQUEST
WRITE REQUEST REQUIRING CHANGE TO ALLOCATION
460
PROCESS 3
WRITE LOCK
430

FIG. 4A

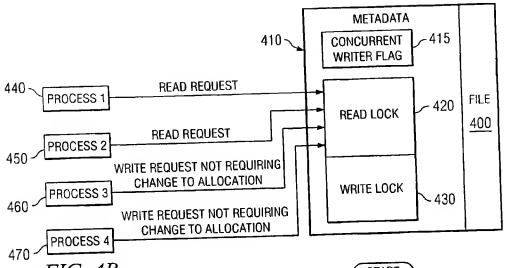

METADATA
410
CONCURRENT WRITER FLAG
415
440
PROCESS 1
READ REQUEST
READ LOCK
420
FILE
400
450
PROCESS 2
READ REQUEST
WRITE REQUEST NOT REQUIRING CHANGE TO ALLOCATION
460
PROCESS 3
WRITE LOCK
430
WRITE REQUEST NOT REQUIRING CHANGE TO ALLOCATION
470
PROCESS 4

FIG. 4B

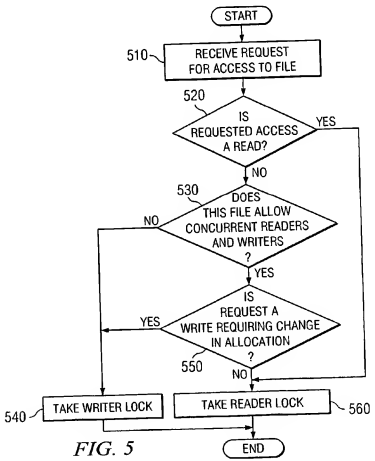

START
510
RECEIVE REQUEST FOR ACCESS TO FILE
520
IS REQUESTED ACCESS A READ?
YES
NO
530
DOES THIS FILE ALLOW CONCURRENT READERS AND WRITERS ?
NO
YES
IS REQUEST A WRITE REQUIRING CHANGE IN ALLOCATION ?
YES
550
NO
540
TAKE WRITER LOCK
TAKE READER LOCK
560
END

FIG. 5

METHOD AND APPARATUS FOR A MULTIPLE CONCURRENT WRITER FILE SYSTEM

BACKGROUND OF THE INVENTION

[0001] 1. Technical Field

[0002] The present invention is generally directed to an improved file system for a data processing system. More specifically, the present invention is directed to a local file system that permits multiple concurrent readers and writers.

[0003] 2. Description of Related Art

[0004] A file system is a computer program that allows other application programs to store and retrieve data on media such as disk drives. A file is a named collection of related information that is recorded on a storage medium, e.g., a magnetic disk. The file system allows application programs to create files, give them names, store (or write) data into them, to read data from them, delete them, and perform other operations on them. In general, a file structure is the organization of data on the disk drives. In addition to the file data itself, the file structure contains metadata: a directory that maps file names to the corresponding files, file metadata that contains information about the file, most importantly the location of the file data on the disk (i.e. which disk blocks hold the file data), an allocation map that records which disk blocks are currently in use to store metadata and file data, and a superblock that contains overall information about the file structure (e.g., the locations of the directory, allocation map, and other metadata structures).

[0005] File systems may be localized, such as a file system for a particular computing device, or distributed such that a plurality of computing devices have access to shared storage, e.g., a shared disk file system. In both cases, it is important to ensure the integrity of the file structure accessed by the file system so that corruption of data is not permitted. This is typically performed by governing the computing devices and/or applications that may read or write to the files of the file structure.

[0006] Consider a file structure stored on N disks, D0, D1, . . . , DN−1. Each disk block in the file structure is identified by a pair (i,j), e.g., (5, 254) identifies the 254^{th} block on disk D5. The allocation map is typically stored in an array A, where the value of element $A(i,j)$ denotes the allocation state (allocated/free) of disk block (i,j).

[0007] The allocation map is typically stored on disk as part of the file structure, residing in one or more disk blocks. Conventionally, $A(i,j)$ is the kth sequential element in the map, where $k=iM+j$, and M is some constant greater than the largest block number on any disk.

[0008] To find a free block of disk space, the file system reads a block of A into a memory buffer and searches the buffer to find an element $(A(i,j)$ whose value indicates that the corresponding block (i,j) is free. Before using block (i,j), the file system updates the value of $A(i,j)$ in the buffer to indicate that the state of the block (i,j) is allocated, and writes the buffer back to disk. To free a block (i,j) that is no long needed, the file system reads the block containing $A(i,j)$ into a buffer, updates the value of $A(i,j)$ to denote that block (i,j) is free, and writes the block from the buffer back to disk.

[0009] If the nodes comprising a shared disk file system, or a plurality of applications on a single computing device, do not properly synchronize their access to the shared storage, they may corrupt the file structure. This applies in particular to the allocation map. To illustrate this, consider the process of allocating a free block described above. Suppose two nodes simultaneously attempt to allocate a block. In the process of doing this, they could both read the same allocation map block, both find the same element $A(i,j)$ describing free block (i,j), both update $A(i,j)$ to show block (i,j) as allocated, both write the block back to disk, and both proceed to use block (i,j) for different purposes, thus violating the integrity of the file structure.

[0010] A more subtle but just as serious problem occurs even if the nodes simultaneously allocate different blocks X and Y, if A(X) and A(Y) arc both contained in the same map block. In this case, the first node sets A(X) to allocated, the second node sets A(Y) to allocated, and both simultaneously write their buffered copies of the map block to disk. Depending on which write is done first, either block X or Y will appear free in the map on the disk. If, for example, the second node's write is executed after the first node's write, block X will be free in the map on disk. The first node will proceed to use block X (e.g., to store a data block on a file), but at some time later another node could allocate block X for some other purpose, again with the result of violating the integrity of the file structure.

[0011] In order to ensure the integrity of the file structure, many file systems make use of an integrity manager or concurrency management mechanism that determines how to govern reads and writes to the storage device. The most widely used mechanism is a locking mechanism in which processes must obtain a lock on a block of data in order to access the block of data. For example, a block of data may have a read lock and a write lock. Any number of processes may obtain the read lock concurrently and thus, be able to read the data in the block at approximately the same time. However, only one process may obtain the write lock at any one time. Thus, multiple concurrent readers are possible but only one writer is permitted at any one time. This ensures that two or more processes cannot write to the same block of data at the same time, such as in the situation previously discussed.

[0012] Some computer applications also provide for their own serialization or locking of blocks of data. For example, databases typically include integrity management mechanisms for ensuring that the integrity of the records within the database is maintained. These application based integrity management mechanisms manage reads and writes to records of the database so that the database is not corrupted.

[0013] An example of such an integrity management mechanism is the two-phase commit. In the two-phase commit, a prepare phase is followed by a commit phase. In the prepare phase, a global coordinator (initiating database) requests that all participants (distributed databases) agree to commit or rollback a transaction. In the subsequent commit phase, all participants respond to the coordinator that they are prepared and then the coordinator requests all nodes to commit the transaction. If all participants cannot prepare or there is a system component failure, the coordinator asks all databases to rollback the transaction.

[0014] In situations where an application, such as a database, provides for its own serialization or locking, there is no need for the file system to limit the number of concurrent

writers to a single writer in order to avoid corruption of the file structure. In fact, in some situations, the potential speed at which the application may execute is impaired by the limitations of the file system. Thus, it would be beneficial to remove the limitations of the file system with regard to concurrent writers when the file in question is associated with an application having its own serialization or locking mechanisms.

SUMMARY OF THE INVENTION

[0015] The present invention provides a method and apparatus for a multiple concurrent reader/writer file system. With the method and apparatus of the present invention, the metadata of a file includes a read lock, a write lock, and a concurrent writer flag. If the concurrent writer flag is set, the file allows for multiple writers. In other words, multiple processes may write to the same block of data within the file at approximately the same time as long as they are not changing the allocation of the block of data, i.e. either allocating the block, deallocating the block of data, or changing the size of the block of data.

[0016] With the method and apparatus of the present invention, when an access request, e.g., a write or a read operation, is received for one or more data blocks of a file, a determination is first made as to whether the access request is a read request. If the access request is a read request, the reader lock of the file is obtained by the process sending the access request. Any number of processes may acquire the reader lock of a file at approximately the same time such that multiple concurrent readers are allowed.

[0017] If the access request is not a read access request, then the access request is determined to be a write access request. A determination is made as to whether the file permits multiple concurrent writers by determining the value of the concurrent writer flag in the metadata for the file. If the concurrent writer flag is set, then the file permits multiple concurrent writers. If the concurrent writer flag is not set, then the file does not permit multiple concurrent writers. If it is determined that multiple concurrent writers is not permitted, i.e. the concurrent writers flag is not set, then the process must obtain the writer lock to gain access to the file. Only one process may acquire the write lock at a time and thus, any subsequent process requesting write access to the file and needing to obtain the write lock will spin on the lock until it is released by the process that currently has acquired it. This also prevents readers from accessing the file. Thus, while there is a reader lock writers will spin on the lock and while there is a writer lock readers will spin on the lock.

[0018] If the file permits concurrent writers, i.e. the concurrent writer flag is set, then a determination is made as to whether the write access request is a write access request that intends to change the allocation of one or more blocks of the file. That is, if the write access request will result in a change in the size of the file either by allocating new data blocks to the file, deallocating existing blocks in the file, or changing the size of the existing blocks. If the write access request is one that will require or result in a change to the allocation of the data blocks of the file, then the write lock must be acquired by this process.

[0019] One situation in which a write access request will change the allocation of the data blocks of the file is when a file is extended, i.e. the request is a request to write to an offset that is greater than the current file size. Another situation where a write access request will change the allocation of the data blocks is when the file is truncated. Both of these situations require an update to the metadata structure associated with the file.

[0020] Another situation that results in a change to the metadata structure of the file is when an input/output request on the file violates the alignment or length restrictions of direct input/output. That is, the use of concurrent input/output preferably makes certain alignment and length restrictions that are to be adhered to by the application's I/O requests. By creating file systems with an appropriate block size, e.g., by specifying an aggregate block size equal to 512 kb at file system creation, such applications can benefit from the use of concurrent I/O without any modifications to the applications.

[0021] If the write access request does not require or result in a change in the allocation of data blocks of the file, then the process acquires a read lock of the file and performs its write operations using the read lock. It should be noted that the read lock does not prevent write operations from being performed on the file. Since multiple processes may acquire the read lock on the file at approximately the same time, there may be multiple concurrent readers and writers to the file at approximately the same time as long as the writers are not changing the allocation of the file.

[0022] Because the present invention is intended to be used in conjunction with applications that have their own serialization of changes to data blocks, e.g., a database application, the permitting of multiple writer processes does not degrade the integrity of the file structure. That is, the present invention removes the requirement that the file system ensure integrity by always permitting only one writer process at a time and allows the application to use its serialization mechanisms to govern how changes to blocks of data are to be committed. Only when actual changes to allocations are being made does the file system of the present invention limit changes to allocations to only one writer process at a time.

[0023] These and other features and advantages of the present invention will be described in, or will become apparent to those of ordinary skill in the art in view of, the following detailed description of the preferred embodiments.

BRIEF DESCRIPTION OF THE DRAWINGS

[0024] The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself, however, as well as a preferred mode of use, further objectives and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

[0025] **FIG. 1** is an exemplary diagram of a distributed data processing system in accordance with the present invention;

[0026] **FIG. 2** is an exemplary diagram of a server computing device in which the present invention may be implemented;

[0027] **FIG. 3** is an exemplary diagram of a client computing device in which the present invention may be implemented;

[0028] **FIG. 4A** is an exemplary diagram illustrating the acquiring of locks with regard to a write access request that requires a change in allocation of data blocks for a file in accordance with the present invention;

[0029] **FIG. 4B** is an exemplary diagram illustrating the acquiring of locks with regard to a write access request that does not change the allocation of data blocks for a file in accordance with the present invention; and

[0030] **FIG. 5** is a flowchart outlining an exemplary operation of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

[0031] The present invention provides a method and apparatus for allowing multiple concurrent writer processes to the same file. The present invention may be implemented in a stand alone computing device or in a distributed data processing system. For example, the present invention may be implemented by a server computing device, a client computing device, a stand alone computing device, or a combination of a server computing device and a client computing device. Therefore, a brief description of a distributed data processing system and stand alone computing device are described hereafter in order to provide a context for the operations of the present invention described thereafter.

[0032] With reference now to the figures, **FIG. 1** depicts a pictorial representation of a network of data processing systems in which the present invention may be implemented. Network data processing system **100** is a network of computers in which the present invention may be implemented. Network data processing system **100** contains a network **102**, which is the medium used to provide communications links between various devices and computers connected together within network data processing system **100**. Network **102** may include connections, such as wire, wireless communication links, or fiber optic cables.

[0033] In the depicted example, server **104** is connected to network **102** along with storage unit **106**. In addition, clients **108**, **110**, and **112** are connected to network **102**. These clients **108**, **110**, and **112** may be, for example, personal computers or network computers. In the depicted example, server **104** provides data, such as boot files, operating system images, and applications to clients **108**-**112**. Clients **108**, **110**, and **112** are clients to server **104**. Network data processing system **100** may include additional servers, clients, and other devices not shown. In the depicted example, network data processing system **100** is the Internet with network **102** representing a worldwide collection of networks and gateways that use the Transmission Control Protocol/Internet Protocol (TCP/IP) suite of protocols to communicate with one another. At the heart of the Internet is a backbone of high-speed data communication lines between major nodes or host computers, consisting of thousands of commercial, government, educational and other computer systems that route data and messages. Of course, network data processing system **100** also may be implemented as a number of different types of networks, such as for example, an intranet, a local area network (LAN), or a wide area network (WAN). **FIG. 1** is intended as an example, and not as an architectural limitation for the present invention.

[0034] Referring to **FIG. 2**, a block diagram of a data processing system that may be implemented as a server, such as server **104** in **FIG. 1**, is depicted in accordance with a preferred embodiment of the present invention. Data processing system **200** may be a symmetric multiprocessor (SMP) system including a plurality of processors **202** and **204** connected to system bus **206**. Alternatively, a single processor system may be employed. Also connected to system bus **206** is memory controller/cache **208**, which provides an interface to local memory **209**. I/O bus bridge **210** is connected to system bus **206** and provides an interface to I/O bus **212**. Memory controller/cache **208** and I/O bus bridge **210** may be integrated as depicted.

[0035] Peripheral component interconnect (PCI) bus bridge **214** connected to I/O bus **212** provides an interface to PCI local bus **216**. A number of modems may be connected to PCI local bus **216**. Typical PCI bus implementations will support four PCI expansion slots or add-in connectors. Communications links to clients **108**-**112** in **FIG. 1** may be provided through modem **218** and network adapter **220** connected to PCI local bus **216** through add-in boards.

[0036] Additional PCI bus bridges **222** and **224** provide interfaces for additional PCI local buses **226** and **228**, from which additional modems or network adapters may be supported. In this manner, data processing system **200** allows connections to multiple network computers. A memory-mapped graphics adapter **230** and hard disk **232** may also be connected to I/O bus **212** as depicted, either directly or indirectly.

[0037] Those of ordinary skill in the art will appreciate that the hardware depicted in **FIG. 2** may vary. For example, other peripheral devices, such as optical disk drives and the like, also may be used in addition to or in place of the hardware depicted. The depicted example is not meant to imply architectural limitations with respect to the present invention.

[0038] The data processing system depicted in **FIG. 2** may be, for example, an IBM eServer pSeries system, a product of International Business Machines Corporation in Armonk, N.Y., running the Advanced Interactive Executive (AIX) operating system or LINUX operating system.

[0039] With reference now to **FIG. 3**, a block diagram illustrating a data processing system is depicted in which the present invention may be implemented. Data processing system **300** is an example of a client computer or a stand alone computing device. Data processing system **300** employs a peripheral component interconnect (PCI) local bus architecture. Although the depicted example employs a PCI bus, other bus architectures such as Accelerated Graphics Port (AGP) and Industry Standard Architecture (ISA) may be used. Processor **302** and main memory **304** are connected to PCI local bus **306** through PCI bridge **308**. PCI bridge **308** also may include an integrated memory controller and cache memory for processor **302**. Additional connections to PCI local bus **306** may be made through direct component interconnection or through add-in boards. In the depicted example, local area network (LAN) adapter **310**, SCSI host bus adapter **312**, and expansion bus interface **314** are connected to PCI local bus **306** by direct component connection. In contrast, audio adapter **316**, graphics adapter **318**, and audio/video adapter **319** are connected to PCI local bus **306** by add-in boards inserted into expansion slots.

Expansion bus interface **314** provides a connection for a keyboard and mouse adapter **320**, modem **322**, and additional memory **324**. Small computer system interface (SCSI) host bus adapter **312** provides a connection for hard disk drive **326**, tape drive **328**, and CD-ROM drive **330**. Typical PCI local bus implementations will support three or four PCI expansion slots or add-in connectors.

[0040] An operating system runs on processor **302** and is used to coordinate and provide control of various components within data processing system **300** in **FIG. 3**. The operating system may be a commercially available operating system, such as Windows XP, which is available from Microsoft Corporation. An object oriented programming system such as Java may run in conjunction with the operating system and provide calls to the operating system from Java programs or applications executing on data processing system **300**. "Java" is a trademark of Sun Microsystems, Inc. Instructions for the operating system, the object-oriented operating system, and applications or programs are located on storage devices, such as hard disk drive **326**, and may be loaded into main memory **304** for execution by processor **302**.

[0041] Those of ordinary skill in the art will appreciate that the hardware in **FIG. 3** may vary depending on the implementation. Other internal hardware or peripheral devices, such as flash read-only memory (ROM), equivalent nonvolatile memory, or optical disk drives and the like, may be used in addition to or in place of the hardware depicted in **FIG. 3**. Also, the processes of the present invention may be applied to a multiprocessor data processing system.

[0042] As another example, data processing system **300** may be a stand-alone system configured to be bootable without relying on some type of network communication interfaces As a further example, data processing system **300** may be a personal digital assistant (PDA) device, which is configured with ROM and/or flash ROM in order to provide non-volatile memory for storing operating system files and/or user-generated data.

[0043] The depicted example in **FIG. 3** and above-described examples are not meant to imply architectural limitations. For example, data processing system **300** also may be a notebook computer or hand held computer in addition to taking the form of a PDA. Data processing system **300** also may be a kiosk or a Web appliance.

[0044] As previously mentioned, the present invention provides a method and apparatus for allowing multiple concurrent writer processes to access the same file at approximately the same time. The present invention is preferably implemented in a computing system that employs an application that has its own serialization mechanisms for ensuring the integrity of changes to files. In a preferred embodiment, this application may be a database application such as Oracle and DB2. However, any database application that enforces their own serialization for accesses to shared files can use concurrent I/O, in accordance with the present invention, to reduce CPU consumption and eliminate the overhead of copying data twice, i.e. first between the disk and the file buffer cache, and then from the file buffer cache to the application's buffer.

[0045] The present invention is predicated on the determination that the limits to concurrent write operations enforced by file systems such that only one write operation may be performed at a time on a file is rooted in the desire to avoid two or more processes from changing the allocation of data blocks in the file and thereby corrupting the file structure. Other software mechanisms exist, such as in database applications, for ensuring consistency of the actual data written to the file data blocks, e.g., the two-phase commit. Therefore, the present invention seeks to remove the limitations of existing file systems with regard to write operations that do not change the allocation of data blocks in a file such that multiple concurrent write operations may be performed with the other software application integrity mechanisms governing how these changes to the file are to be implemented.

[0046] With the present invention, write operations that do not require or result in a change to the allocation of data blocks associated with a file may take a reader lock rather than the writer lock. As a result, multiple concurrent write operations may be performed by processes as long as those write operations do not change the allocation of the block of data. If, however, a write operation changes the allocation of a block of data, then the write operation must obtain the writer lock before the operation may be performed. Since only one process may obtain the writer lock at a time, this forces serialization of write operations that change the allocation of data blocks in a file. That is, each write operation that changes an allocation must wait unit the writer lock is released by a process that currently is changing the allocation of data blocks in the file before it can perform its operations. The present invention does not avoid or bypass the file locking, but makes use of the file locks to permit multiple concurrent readers and writers.

[0047] **FIG. 4A** is an exemplary diagram illustrating the acquiring of locks with regard to a write access request that requires a change in allocation of data blocks for a file in accordance with the present invention. As shown in **FIG. 4A**, a file **400** has associated metadata **410** that includes a concurrent writer flag **415**, a read lock **420** and a write lock **430**. The concurrent writer flag **415** may be set by an application that initially creates the file **400** to indicate whether that application permits concurrent writers to the file **400**. With the present invention, only applications that have their own internal serialization or integrity management mechanisms may set the concurrent writer flag **415** such that the file **400** may be accessed by multiple concurrent writers, i.e. processes that are requesting write access to the file **400**. An example of such an application is a database application which includes its own serialization mechanisms for serializing the concurrent writes to data blocks in order to maintain the integrity of the file structure.

[0048] In order for a process to access the file **400**, the process must obtain a lock on the file **400**. If the process wishes to read data from the file **400**, the process may obtain a read lock **420** associated with the file **400**. If the process wishes to write data to the file **400**, the process may have to obtain either the read lock **420** or the write lock **430** depending on the type of write operation being performed.

[0049] If the write operation that is being performed by a process is one that requires or results in a change in the allocation of data blocks to the file **400**, then the process requesting access to the file **400** must obtain the write lock **430**. The access policy associated with the metadata pre-

cludes more than one process from acquiring the write lock **430** at any one time. Thus, if two processes are attempting to write the file **400**, and both processes' write operations require or result in a change to the allocation of data blocks in the file **400**, then only one of these processes will be allowed to proceed by obtaining the write lock **430** while the other must spin on the lock. It should also be noted that readers must also spin while the writer lock is taken and the write lock cannot be taken while there is a reader lock.

[0050] Thus, as shown in **FIG. 4A**, process **1440** and process **2450** send read access requests to the file system requesting access to the file **400** so that they may read data from the file **400**. As a result, each of process **1440** and process **2450** obtain the read lock **420** associated with the file **400**. Process **3460**, however, sends a write access request to the file system requesting access to the file **400** so that the process **460** may write data to the file **400**. This writing of data is determined to require or result in a change in the allocation of data blocks within file **400**.

[0051] As previously mentioned, one situation in which a write access request will change the allocation of the data blocks of the file is when a file is extended, i.e. the request is a request to write to an offset that is greater than the current file size. Another situation where a write access request will change the allocation of the data blocks is when the file is truncated. Both of these situations require an update to the metadata structure associated with the file.

[0052] Another situation that results in a change to the metadata structure of the file is when an input/output request on the file violates the alignment or length restrictions of direct input/output. That is, the use of concurrent input/output preferably makes certain alignment and length restrictions that are to be adhered to by the application's I/O requests. By creating file systems with an appropriate block size, e.g., by specifying an aggregate block size equal to 512 kb at file system creation, such applications can benefit from the use of concurrent I/O without any modifications to the applications.

[0053] As a result of determining that the Process **3460** requires a change in the allocation data blocks within the file **400**, the process **460** must obtain the write lock **430** in order to perform its write operations to data blocks of the file **400**. If the process **460** is unable to acquire the write lock **430** immediately, the process **460** may spin on the write lock **430** until it is released by the process that currently has the write lock **430**.

[0054] With the present invention, if the write operation of a process will not require or result in a change in the allocation of the data blocks in the file **400**, then the process may obtain the read lock **420** rather than being forced to obtain the write lock **430**. That is, the present invention differentiates between two different types of write accesses, a write that will change the allocation of data blocks in the file **400** and a write that will not change the allocation of data blocks in the file **400**.

[0055] **FIG. 4B** is an exemplary diagram illustrating the acquiring of locks with regard to a write access request that does not change the allocation of data blocks for a file in accordance with the present invention. As illustrated in **FIG. 4B**, the processes **440** and **450** send read access requests to the file system requesting access to the file **400** to read data from the file **400**. These processes acquire the read lock **420** and are able to concurrently perform read operations on the data in the file **400**.

[0056] The processes **460** and **470** submit write access requests to the file system requesting access to the file **400** to write data to the file **400**. The write operations that processes **460** and **470** are intending to perform are determined to be of a type that does not require or result in a change to the allocation of data blocks in file **400**. Since the write operations do not change the allocation of data blocks in the file **400**, the processes **460** and **470** are permitted to acquire the read lock **420** and thus, are able to concurrently write data to the file **400**. Software based mechanisms, such as database application serialization mechanisms, are utilized to determine how the concurrent write operations are to be serialized such that file structure integrity is maintained.

[0057] Thus, the present invention provides a mechanism for eliminating the bottleneck to performance found in the access policy of conventional file systems with regard to permitting only a single writer to a file at any one time. With the present invention, this limitation is lifted with regard to write operations that do not require or result in a change in the allocation of data blocks in the file. As a result, multiple concurrent write operations may be performed without sacrificing the file structure integrity. Existing software based serialization and locking mechanisms associated with an application present on the computing system are utilized to govern how these concurrent write operations are to be reflected in the file structure such that the integrity of the file structure is maintained.

[0058] **FIG. 5** is a flowchart outlining an exemplary operation of the present invention. It will be understood that each block of the flowchart illustration, and combinations of blocks in the flowchart illustration, can be implemented by computer program instructions. These computer program instructions may be provided to a processor or other programmable data processing apparatus to produce a machine, such that the instructions which execute on the processor or other programmable data processing apparatus create means for implementing the functions specified in the flowchart block or blocks. These computer program instructions may also be stored in a computer-readable memory or storage medium that can direct a processor or other programmable data processing apparatus to function in a particular manner, such that the instructions stored in the computer-readable memory or storage medium produce an article of manufacture including instruction means which implement the functions specified in the flowchart block or blocks.

[0059] Accordingly, blocks of the flowchart illustration support combinations of means for performing the specified functions, combinations of steps for performing the specified functions and program instruction means for performing the specified functions. It will also be understood that each block of the flowchart illustration, and combinations of blocks in the flowchart illustration, can be implemented by special purpose hardware-based computer systems which perform the specified functions or steps, or by combinations of special purpose hardware and computer instructions.

[0060] As shown in **FIG. 5**, the operation starts by receiving a request for access to a file (**step 510**). A determination is made as to whether this access request is a read access

request (step **520**). If so, the reader lock is taken (step **560**). If the request is not a read request then it is determined that the request is a write access request.

[**0061**] If the access request is not a read access request, a determination is made as to whether the file to which access is requested allows concurrent readers and writers (step **530**). As mentioned above, this may involve determining the value of a concurrent writer flag in the metadata of the file, for example. If the file does not permit concurrent writers, the writer lock is taken (step **540**). This assumes that the writer lock is available and has not been acquired by another process. If the writer lock is already acquired by another process, the current process may spin on the lock until it is released so that the current process may acquire it. As mentioned above, only one process may acquire the writer lock at any one time and thus, no other processes that are attempting to perform a write to the file will be able to perform their operation until after the writer lock is released.

[**0062**] If the file does allow multiple concurrent writers, then a determination is made as to whether the write request is one that will require or result in a change in the allocation of data blocks in the file (step **550**). If so, the writer lock is acquired (step **540**) as discussed above. Otherwise, if the write request is one that will not require or result in a change in the allocation of data blocks in the file, then a reader lock may be acquired by the process submitting the write request (step **560**). As previously mentioned, multiple processes may acquire the reader lock on the file and thereby access the file concurrently. With the present invention, since write requests that do not change the allocation of data blocks of a file may acquire this lock, multiple concurrent writers to the file are possible. The present invention allows the serialization mechanisms of the applications of the computing device, e.g., the database application, to govern how changes to the file are to be committed. Thus, the file system of the present invention only limits processes from writing to a file concurrently when the write operations would result in a change in the allocation of data blocks of the file.

[**0063**] It is important to note that while the present invention has been described in the context of a fully functioning data processing system, those of ordinary skill in the art will appreciate that the processes of the present invention are capable of being distributed in the form of a computer readable medium of instructions and a variety of forms and that the present invention applies equally regardless of the particular type of signal bearing media actually used to carry out the distribution. Examples of computer readable media include recordable-type media, such as a floppy disk, a hard disk drive, a RAM, CD-ROMs, DVD-ROMs, and transmission-type media, such as digital and analog communications links, wired or wireless communications links using transmission forms, such as, for example, radio frequency and light wave transmissions. The computer readable media may take the form of coded formats that are decoded for actual use in a particular data processing system.

[**0064**] The description of the present invention has been presented for purposes of illustration and description, and is not intended to be exhaustive or limited to the invention in the form disclosed. Many modifications and variations will be apparent to those of ordinary skill in the art. The embodiment was chosen and described in order to best explain the principles of the invention, the practical application, and to enable others of ordinary skill in the art to understand the invention for various embodiments with various modifications as are suited to the particular use contemplated.

What is claimed is:

1. A method of providing write access to a file, comprising:

receiving a write access request from a process for write access to the file;

determining if a write operation associated with the write access request results in a change to an allocation of data blocks in the file; and

permitting the process to obtain a read lock associated with the file to perform the write operation if the write operation does not result in a change to the allocation of data blocks in the file.

2. The method of claim 1, further comprising:

requiring that the process obtain a write lock associated with the file to perform the write operation if the write operation results in a change to the allocation of data blocks in the file.

3. The method of claim 1, wherein multiple processes may have concurrent access to the file by obtaining a read lock associated with the file.

4. The method of claim 2, wherein only one process may obtain the write lock at a time.

5. The method of claim 1, wherein the process performs the write operation to the file concurrently with another write operation to the file from another process.

6. The method of claim 1, wherein determining if the write operation results in a change to an allocation of data blocks in the file includes determining if the write operation is to an offset that is greater than a current file size.

7. The method of claim 1, wherein determining if the write operation results in a change to an allocation of data blocks in the file includes determining if the write operation is to truncate the file.

8. A computer program product in a computer readable medium for providing write access to a file, comprising:

first instructions for receiving a write access request from a process for write access to the file;

second instructions for determining if a write operation associated with the write access request results in a change to an allocation of data blocks in the file; and

third instructions for permitting the process to obtain a read lock associated with the file to perform the write operation if the write operation does not result in a change to the allocation of data blocks in the file.

9. The computer program product of claim 8, further comprising:

fourth instructions for requiring that the process obtain a write lock associated with the file to perform the write operation if the write operation results in a change to the allocation of data blocks in the file.

10. The computer program product of claim 8, wherein multiple processes may have concurrent access to the file by obtaining a read lock associated with the file.

11. The computer program product of claim 9, wherein only one process may obtain the write lock at a time.

12. The computer program product of claim 8, wherein the process performs the write operation to the file concurrently with another write operation to the file from another process.

13. The computer program product of claim 8, wherein the second instructions for determining if the write operation results in a change to an allocation of data blocks in the file include instructions for determining if the write operation is to an offset that is greater than a current file size.

14. The computer program product of claim 8, wherein the second instructions for determining if the write operation results in a change to an allocation of data blocks in the file include instructions for determining if the write operation is to truncate the file.

15. An apparatus for providing write access to a file, comprising:

means for receiving a write access request from a process for write access to the file;

means for determining if a write operation associated with the write access request results in a change to an allocation of data blocks in the file; and

means for permitting the process to obtain a read lock associated with the file to perform the write operation if the write operation does not result in a change to the allocation of data blocks in the file.

16. The apparatus of claim 15, further comprising:

means for requiring that the process obtain a write lock associated with the file to perform the write operation if the write operation results in a change to the allocation of data blocks in the file.

17. The apparatus of claim 15, wherein multiple processes may have concurrent access to the file by obtaining a read lock associated with the file.

18. The apparatus of claim 16, wherein only one process may obtain the write lock at a time.

19. The apparatus of claim 15, wherein the process performs the write operation to the file concurrently with another write operation to the file from another process.

20. The apparatus of claim 15, wherein the means for determining if the write operation results in a change to an allocation of data blocks in the file includes means for determining if the write operation is to an offset that is greater than a current file size.

21. The apparatus of claim 15, wherein the means for determining if the write operation results in a change to an allocation of data blocks in the file includes means for determining if the write operation is to truncate the file.

* * * * *

US005163131A

United States Patent [19]

Row et al.

[11] Patent Number: 5,163,131

[45] Date of Patent: Nov. 10, 1992

[54] PARALLEL I/O NETWORK FILE SERVER ARCHITECTURE

[75] Inventors: Edward J. Row, Mountain View; Laurence B. Boucher, Saratoga; William M. Pitts, Los Altos; Stephen E. Blightman, San Jose, all of Calif.

[73] Assignee: Auspex Systems, Inc., Santa Clara, Calif.

[21] Appl. No.: 404,959

[22] Filed: Sep. 8, 1989

[51] Int. Cl.[5] G06F 15/16; G06F 13/00

[52] U.S. Cl. 395/200; 364/DIG. 1; 364/242.4; 364/228.3; 364/284; 364/284.4; 364/243.4; 364/230

[58] Field of Search ... 364/200 MS File, 900 MS File; 395/200, 650

[56] References Cited

U.S. PATENT DOCUMENTS

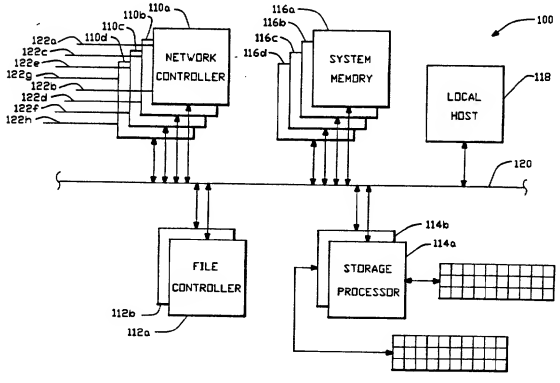
4,527,232	7/1985	Bechtolsheim	364/200
4,550,368	10/1985	Bechtolsheim	364/200
4,710,868	12/1987	Cocke et al.	364/200
4,719,569	1/1988	Ludemann et al.	364/200
4,803,621	2/1989	Kelly	364/200
4,819,159	4/1989	Shipley et al.	364/200
4,887,204	12/1989	Johnson et al.	364/200
4,897,781	1/1990	Chang et al.	364/200

Primary Examiner—Kevin A. Kriess
Attorney, Agent, or Firm—Fliesler, Dubb, Meyer & Lovejoy

[57] ABSTRACT

A file server architecture is disclosed, comprising as separate processors, a network controller unit, a file controller unit and a storage processor unit. These units incorporate their own processors, and operate in parallel with a local Unix host processor. All networks are connected to the network controller unit, which performs all protocol processing up through the NFS layer. The virtual file system is implemented in the file control unit, and the storage processor provides high-speed multiplexed access to an array of mass storage devices. The file controller unit control file information caching through its own local cache buffer, and controls disk data caching through a large system memory which is accessible on a bus by any of the processors.

67 Claims, 12 Drawing Sheets



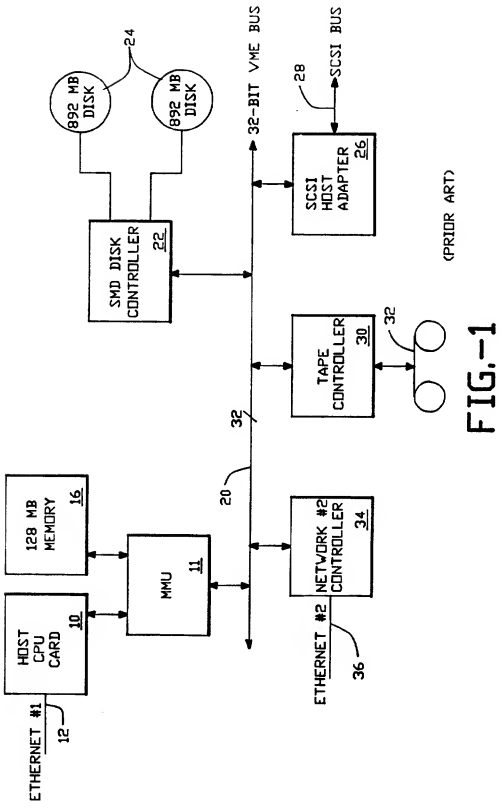

ETHERNET #1
12
HOST CPU CARD
10
128 MB MEMORY
16
MMU
11
20
32
32-BIT VME BUS
NETWORK #2 CONTROLLER
34
ETHERNET #2
36
TAPE CONTROLLER
30
32
SMD DISK CONTROLLER
22
892 MB DISK
892 MB DISK
24
SCSI HOST ADAPTER
26
28
SCSI BUS
(PRIOR ART)

FIG.-1

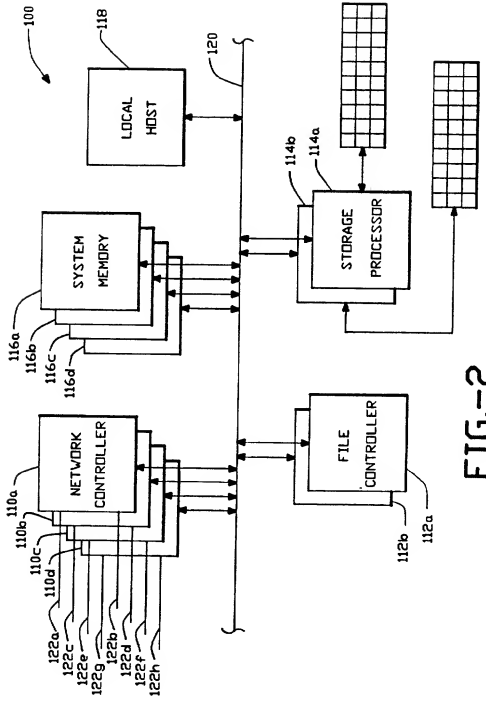

100
118
LOCAL HOST
120
114b
114a
STORAGE PROCESSOR
SYSTEM MEMORY
116a
116b
116c
116d
NETWORK CONTROLLER
110a
110b
110c
110d
122a
122c
122e
122g
122b
122d
122f
122h
FILE CONTROLLER
112b
112a

FIG.-2

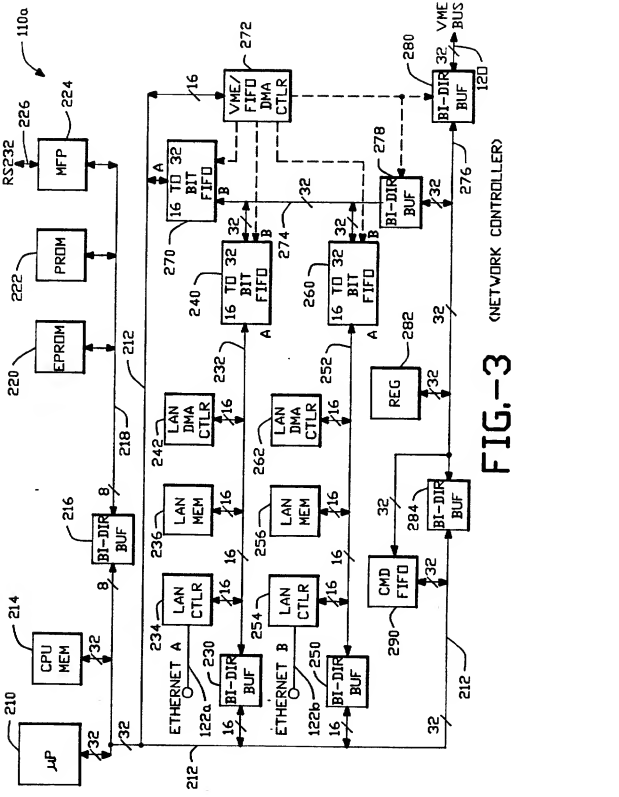

110a
μP
210
CPU MEM
214
MFP
224
RS232
226
PROM
222
EPROM
220
212
218
216
BI-DIR BUF
8
32
LAN CTLR
234
LAN MEM
236
LAN DMA CTLR
242
16 TO 32 BIT FIFO
240
16 TO 32 BIT FIFO
270
16 TO 32 BIT FIFO
260
VME/ FIFO DMA CTLR
272
BI-DIR BUF
278
BI-DIR BUF
280
VME BUS
120
276
274
232
REG
282
CMD FIFO
290
BI-DIR BUF
284
ETHERNET A
122a
230
ETHERNET B
122b
250
LAN CTLR
254
LAN MEM
256
LAN DMA CTLR
262
252
(NETWORK CONTROLLER)

FIG.-3

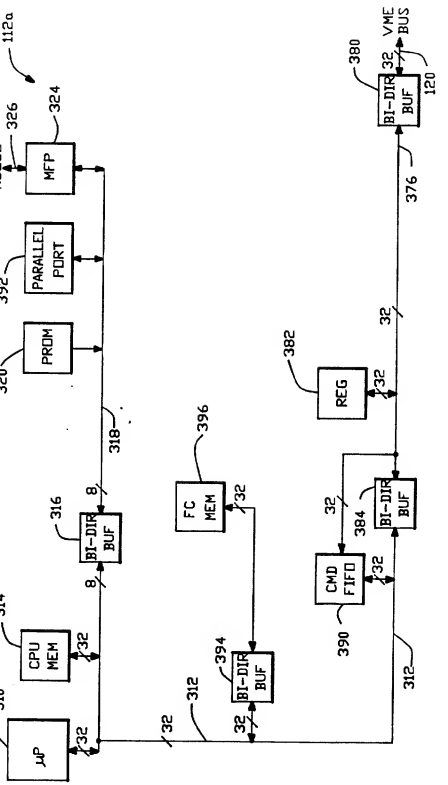

112a
326
324
RS232
MFP
392
PARALLEL PORT
320
PROM
318
316
BI-DIR BUF
8
32
314
CPU MEM
310
μP
312
396
FC MEM
394
BI-DIR BUF
382
REG
390
CMD FIFO
384
BI-DIR BUF
380
BI-DIR BUF
VME BUS
120
376
(FILE CONTROLLER)

FIG.-4

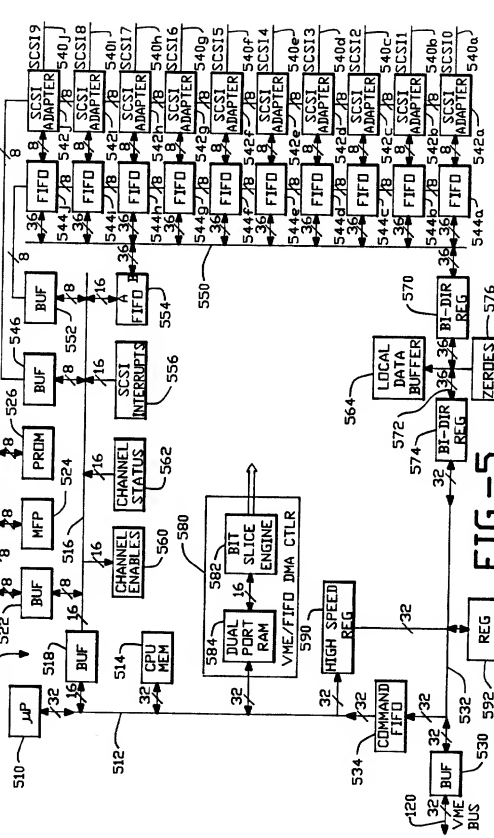

μP
BUF
CPU MEM
DUAL PORT RAM
BIT SLICE ENGINE
VME/FIFO DMA CTLR
HIGH SPEED REG
COMMAND FIFO
REG
BUF
VME BUS
MFP
PROM
CHANNEL ENABLES
CHANNEL STATUS
SCSI INTERRUPTS
A FIFO
BUF
FIFO
SCSI ADAPTER
LOCAL DATA BUFFER
BI-DIR REG
ZERDES

FIG.-5

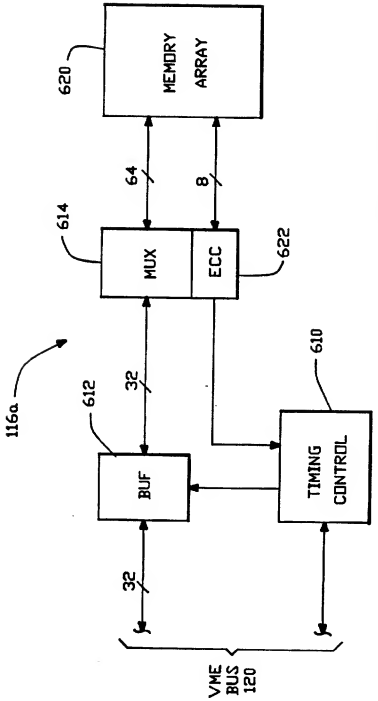

620
MEMORY ARRAY
64
8
614
MUX
ECC
622
(SYSTEM MEMORY)
116a
612
32
610
BUF
TIMING CONTROL
32
VME BUS 120

FIG.-6

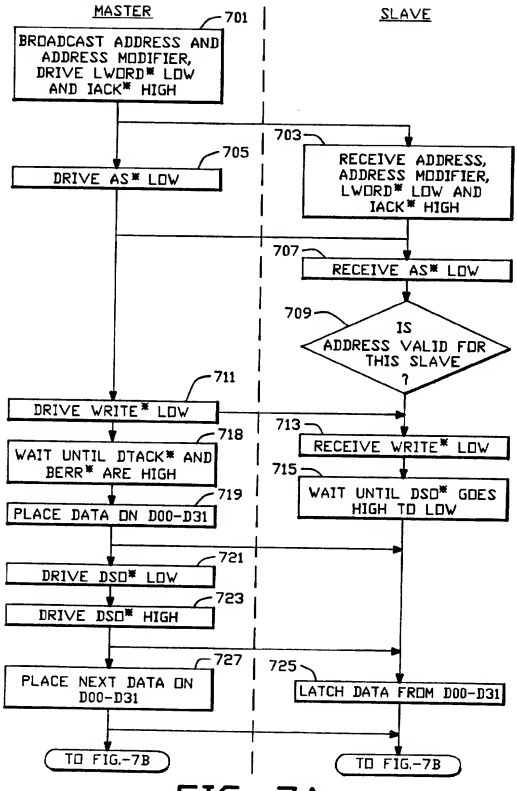

MASTER
SLAVE
701
BROADCAST ADDRESS AND ADDRESS MODIFIER, DRIVE LWORD* LOW AND IACK* HIGH
703
RECEIVE ADDRESS, ADDRESS MODIFIER, LWORD* LOW AND IACK* HIGH
705
DRIVE AS* LOW
707
RECEIVE AS* LOW
709
IS ADDRESS VALID FOR THIS SLAVE ?
711
DRIVE WRITE* LOW
713
RECEIVE WRITE* LOW
718
WAIT UNTIL DTACK* AND BERR* ARE HIGH
715
WAIT UNTIL DSO* GOES HIGH TO LOW
719
PLACE DATA ON DOO-D31
721
DRIVE DSO* LOW
723
DRIVE DSO* HIGH
727
PLACE NEXT DATA ON DOO-D31
725
LATCH DATA FROM DOO-D31
TO FIG.-7B
TO FIG.-7B

FIG.-7A

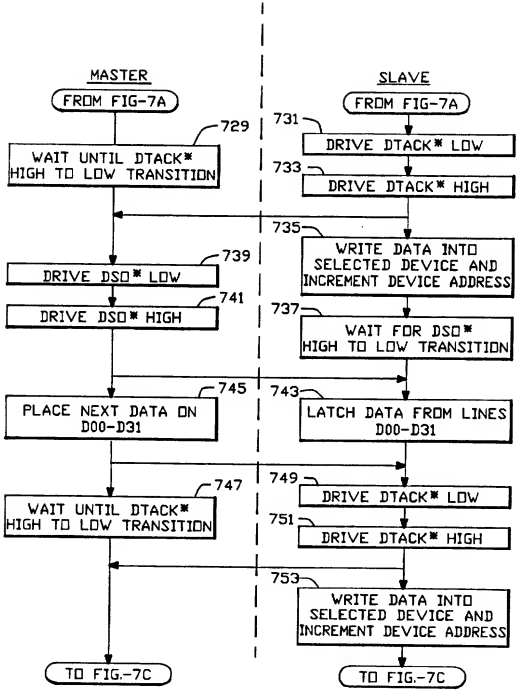

MASTER
FROM FIG-7A
729
WAIT UNTIL DTACK* HIGH TO LOW TRANSITION
739
DRIVE DSO* LOW
741
DRIVE DSO* HIGH
745
PLACE NEXT DATA ON D00-D31
747
WAIT UNTIL DTACK* HIGH TO LOW TRANSITION
TO FIG.-7C
SLAVE
FROM FIG-7A
731
DRIVE DTACK* LOW
733
DRIVE DTACK* HIGH
735
WRITE DATA INTO SELECTED DEVICE AND INCREMENT DEVICE ADDRESS
737
WAIT FOR DSO* HIGH TO LOW TRANSITION
743
LATCH DATA FROM LINES D00-D31
749
DRIVE DTACK* LOW
751
DRIVE DTACK* HIGH
753
WRITE DATA INTO SELECTED DEVICE AND INCREMENT DEVICE ADDRESS
TO FIG.-7C

FIG.-7B

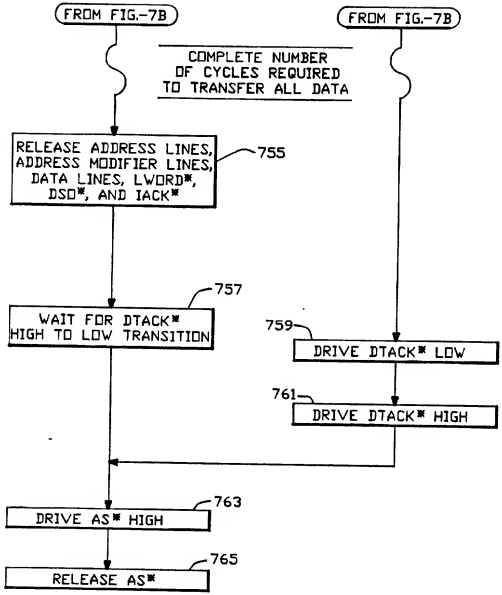

FROM FIG.-7B
FROM FIG.-7B
COMPLETE NUMBER OF CYCLES REQUIRED TO TRANSFER ALL DATA
RELEASE ADDRESS LINES, ADDRESS MODIFIER LINES, DATA LINES, LWORD*, DS0*, AND IACK*
755
WAIT FOR DTACK* HIGH TO LOW TRANSITION
757
759
DRIVE DTACK* LOW
761
DRIVE DTACK* HIGH
DRIVE AS* HIGH
763
RELEASE AS*
765

FIG.-7C

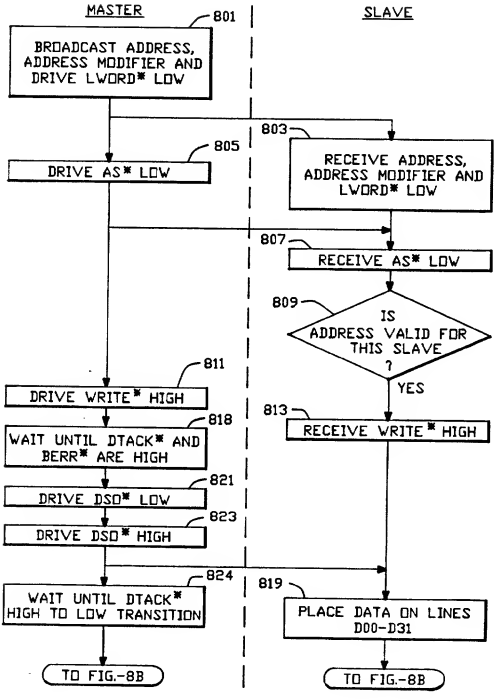

MASTER
SLAVE
801
BROADCAST ADDRESS, ADDRESS MODIFIER AND DRIVE LWORD* LOW
803
RECEIVE ADDRESS, ADDRESS MODIFIER AND LWORD* LOW
805
DRIVE AS* LOW
807
RECEIVE AS* LOW
809
IS ADDRESS VALID FOR THIS SLAVE ?
YES
811
DRIVE WRITE* HIGH
813
RECEIVE WRITE* HIGH
818
WAIT UNTIL DTACK* AND BERR* ARE HIGH
821
DRIVE DSO* LOW
823
DRIVE DSO* HIGH
824
WAIT UNTIL DTACK* HIGH TO LOW TRANSITION
819
PLACE DATA ON LINES D00-D31
TO FIG.-8B
TO FIG.-8B

FIG.-8A

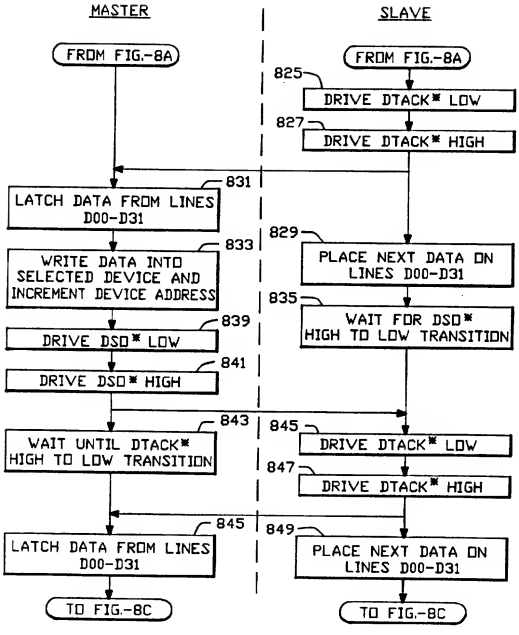

MASTER
SLAVE
FROM FIG.-8A
FROM FIG.-8A
825
DRIVE DTACK* LOW
827
DRIVE DTACK* HIGH
831
LATCH DATA FROM LINES D00-D31
829
PLACE NEXT DATA ON LINES D00-D31
833
WRITE DATA INTO SELECTED DEVICE AND INCREMENT DEVICE ADDRESS
835
WAIT FOR DS0* HIGH TO LOW TRANSITION
839
DRIVE DS0* LOW
841
DRIVE DS0* HIGH
843
WAIT UNTIL DTACK* HIGH TO LOW TRANSITION
845
DRIVE DTACK* LOW
847
DRIVE DTACK* HIGH
845
LATCH DATA FROM LINES D00-D31
849
PLACE NEXT DATA ON LINES D00-D31
TO FIG.-8C
TO FIG.-8C

FIG.-8B

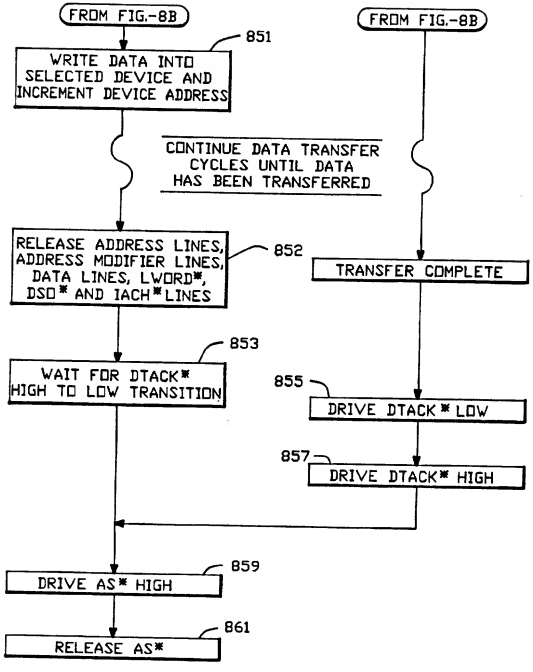

FROM FIG.-8B
FROM FIG.-8B
851
WRITE DATA INTO SELECTED DEVICE AND INCREMENT DEVICE ADDRESS
CONTINUE DATA TRANSFER CYCLES UNTIL DATA HAS BEEN TRANSFERRED
RELEASE ADDRESS LINES, ADDRESS MODIFIER LINES, DATA LINES, LWORD*, DSO* AND IACH* LINES
852
TRANSFER COMPLETE
853
WAIT FOR DTACK* HIGH TO LOW TRANSITION
855
DRIVE DTACK* LOW
857
DRIVE DTACK* HIGH
859
DRIVE AS* HIGH
861
RELEASE AS*

FIG.-8C

PARALLEL I/O NETWORK FILE SERVER ARCHITECTURE

CROSS-REFERENCE TO RELATED APPLICATIONS

The present application is related to the following U.S. patent applications, all filed concurrently herewith:

1. MULTIPLE FACILITY OPERATING SYSTEM ARCHITECTURE, invented by David Hitz, Allan Schwartz, James Lau and Guy Harris;

2. ENHANCED VMEBUS PROTOCOL UTILIZING PSEUDOSYNCHRONOUS HANDSHAKING AND BLOCK MODE DATA TRANSFER, invented by Daryl Starr; and

3. BUS LOCKING FIFO MULTI-PROCESSOR COMMUNICATIONS SYSTEM UTILIZING PSEUDOSYNCHRONOUS HANDSHAKING AND BLOCK MODE DATA TRANSFER invented by Daryl D. Starr, William Pitts and Stephen Blightman.

The above applications are all assigned to the assignee of the present invention and are all expressly incorporated herein by reference.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The invention relates to computer data networks, and more particularly, to network file server architectures for computer networks.

2. Description of the Related Art

Over the past ten years, remarkable increases in hardware price/performance ratios have caused a startling shift in both technical and office computing environments. Distributed workstation-server networks are displacing the once pervasive dumb terminal attached to mainframe or minicomputer. To date, however, network I/O limitations have constrained the potential performance available to workstation users. This situation has developed in part because dramatic jumps in microprocessor performance have exceeded increases in network I/O performance.

In a computer network, individual user workstations are referred to as clients, and shared resources for filing, printing, data storage and wide-area communications are referred to as servers. Clients and servers are all considered nodes of a network. Client nodes use standard communications protocols to exchange service requests and responses with server nodes.

Present-day network clients and servers usually run the DOS, MacIntosh OS, OS/2, or Unix operating systems. Local networks are usually Ethernet or Token Ring at the high end, Arcnet in the midrange, or LocalTalk or StarLAN at the low end. The client-server communication protocols are fairly strictly dictated by the operating system environment—usually one of several proprietary schemes for PCs (NetWare, 3Plus, Vines, LANManager, LANServer); AppleTalk for MacIntoshes; and TCP/IP with NFS or RFS for Unix. These protocols are all well-known in the industry.

Unix client nodes typically feature a 16- or 32-bit microprocessor with 1-8 MB of primary memory, a 640×1024 pixel display, and a built-in network interface. A 40-100 MB local disk is often optional. Low-end examples are 80286-based PCs or 68000-based MacIntosh I's; mid-range machines include 80386 PCs, MacIntosh II's, and 680X0-based Unix workstations; high-end machines include RISC-based DEC, HP, and Sun Unix workstations. Servers are typically nothing more than repackaged client nodes, configured in 19-inch racks rather than desk sideboxes. The extra space of a 19-inch rack is used for additional backplane slots, disk or tape drives, and power supplies.

Driven by RISC and CISC microprocessor developments, client workstation performance has increased by more than a factor of ten in the last few years. Concurrently, these extremely fast clients have also gained an appetite for data that remote servers are unable to satisfy. Because the I/O shortfall is most dramatic in the Unix environment, the description of the preferred embodiment of the present invention will focus on Unix file servers. The architectural principles that solve the Unix server I/O problem, however, extend easily to server performance bottlenecks in other operating system environments as well. Similarly, the description of the preferred embodiment will focus on Ethernet implementations, though the principles extend easily to other types of networks.

In most Unix environments, clients and servers exchange file data using the Network File System ("NFS"), a standard promulgated by Sun Microsystems and now widely adopted by the Unix community. NFS is defined in a document entitled, "NFS: Network File System Protocol Specification," Request For Comments (RFC) 1094, by Sun Microsystems, Inc. (March 1989). This document is incorporated herein by reference in its entirety.

While simple and reliable, NFS is not optimal. Clients using NFS place considerable demands upon both networks and NFS servers supplying clients with NFS data. This demand is particularly acute for so-called diskless clients that have no local disks and therefore depend on a file server for application binaries and virtual memory paging as well as data. For these Unix client-server configurations, the ten-to-one increase in client power has not been matched by a ten-to-one increase in Ethernet capacity, in disk speed, or server disk-to-network I/O throughput.

The result is that the number of diskless clients that a single modern high-end server can adequately support has dropped to between 5-10, depending on client power and application workload. For clients containing small local disks for applications and paging, referred to as dataless clients, the client-to-server ratio is about twice this, or between 10-20.

Such low client/server ratios cause piecewise network configurations in which each local Ethernet contains isolated traffic for its own 5-10 (diskless) clients and dedicated server. For overall connectivity, these local networks are usually joined together with an Ethernet backbone or, in the future, with an FDDI backbone. These backbones are typically connected to the local networks either by IP routers or MAC-level bridges, coupling the local networks together directly, or by a second server functioning as a network interface, coupling servers for all the local networks together.

In addition to performance considerations, the low client-to-server ratio creates computing problems in several additional ways:

1. Sharing

Development groups of more than 50-people cannot share the same server, and thus cannot easily share files without file replication and manual, multi-server up-

dates. Bridges or routers are a partial solution but inflict a performance penalty due to more network hops.

2. Administration

System administrators must maintain many limited-capacity servers rather than a few more substantial servers. This burden includes network administration, hardware maintenance, and user account administration.

3. File System Backup

System administrators or operators must conduct multiple file system backups, which can be onerously time consuming tasks. It is also expensive to duplicate backup peripherals on each server (or every few servers if slower network backup is used).

4. Price Per Seat

With only 5-10 clients per server, the cost of the server must be shared by only a small number of users. The real cost of an entry-level Unix workstation is therefore significantly greater, often as much as 140% greater, than the cost of the workstation alone.

The widening I/O gap, as well as administrative and economic considerations, demonstrates a need for higher-performance, larger-capacity Unix file servers. Conversion of a display-less workstation into a server may address disk capacity issues, but does nothing to address fundamental I/O limitations. As an NFS server, the one-time workstation must sustain 5-10 or more times the network, disk, backplane, and file system *throughput* than it was designed to support as a client. Adding larger disks, more network adaptors, extra primary memory, or even a faster processor do not resolve basic architectural I/O constraints; I/O throughput does not increase sufficiently.

Other prior art computer architectures, while not specifically designed as file servers, may potentially be used as such. In one such well-known architecture, a CPU, a memory unit, and two I/O processors are connected to a single bus. One of the I/O processors operates a set of disk drives, and if the architecture is to be used as a server, the other I/O processor would be connected to a network. This architecture is not optimal as a file server, however, at least because the two I/O processors cannot handle network file requests without involving the CPU. All network file requests that are received by the network I/O processor are first transmitted to the CPU, which makes appropriate requests to the disk-I/O processor for satisfaction of the network request.

In another such computer architecture, a disk controller CPU manages access to disk drives, and several other CPUs, three for example, may be clustered around the disk controller CPU. Each of the other CPUs can be connected to its own network. The network CPUs are each connected to the disk controller CPU as well as to each other for interprocessor communication. One of the disadvantages of this computer architecture is that each CPU in the system runs its own complete operating system. Thus, network file server requests must be handled by an operating system which is also heavily loaded with facilities and processes for performing a large number of other, non file-server tasks. Additionally, the interprocessor communication is not optimized for file server type requests.

In yet another computer architecture, a plurality of CPUs, each having its own cache memory for data and instruction storage, are connected to a common bus with a system memory and a disk controller. The disk controller and each of the CPUs have direct memory access to the system memory, and one or more of the CPUs can be connected to a network. This architecture is disadvantageous as a file server because, among other things, both file data and the instructions for the CPUs reside in the same system memory. There will be instances, therefore, in which the CPUs must stop running while they wait for large blocks of file data to be transferred between system memory and the network CPU. Additionally, as with both of the previously described computer architectures, the entire operating system runs on each of the CPUs, including the network CPU.

In yet another type of computer architecture, a large number of CPUs are connected together in a hypercube topology. One of more of these CPUs can be connected to networks, while another can be connected to disk drives. This architecture is also disadvantageous as a file server because, among other things, each processor runs the entire operating system. Interprocessor communication is also not optimal for file server applications.

SUMMARY OF THE INVENTION

The present invention involves a new, server-specific I/O architecture that is optimized for a Unix file server's most common actions—file operations. Roughly stated, the invention involves a file server architecture comprising one or more network controllers, one or more file controllers, one or more storage processors, and a system or buffer memory, all connected over a message passing bus and operating in parallel with the Unix host processor. The network controllers each connect to one or more network, and provide all protocol processing between the network layer data format and an internal file server format for communicating client requests to other processors in the server. Only those data packets which cannot be interpreted by the network controllers, for example client requests to run a client-defined program on the server, are transmitted to the Unix host for processing. Thus the network controllers, file controllers and storage processors contain only small parts of an overall operating system, and each is optimized for the particular type of work to which it is dedicated.

Client requests for file operations are transmitted to one of the file controllers which, independently of the Unix host, manages the virtual file system of a mass storage device which is coupled to the storage processors. The file controllers may also control data buffering between the storage processors and the network controllers, through the system memory. The file controllers preferably each include a local buffer memory for caching file control information, separate from the system memory for caching file data. Additionally, the network controllers, file processors and storage processors are all designed to avoid any instruction fetches from the system memory, instead keeping all instruction memory separate and local. This arrangement eliminates contention on the backplane between microprocessor instruction fetches and transmissions of message and file data.

BRIEF DESCRIPTION OF THE DRAWINGS

The invention will be described with respect to particular embodiments thereof, and reference will be made to the drawings, in which:

FIG. 1 is a block diagram of a prior art file server architecture;

FIG. 2 is a block diagram of a file server architecture according to the invention;

FIG. 3 is a block diagram of one of the network controllers shown in FIG. 2;

FIG. 4 is a block diagram of one of the file controllers shown in FIG. 2;

FIG. 5 is a block diagram of one of the storage processors shown in FIG. 2;

FIG. 6 is a block diagram of one of the system memory cards shown in FIG. 2;

FIGS. 7A–C are a flowchart illustrating the operation of a fast transfer protocol BLOCK WRITE cycle; and

FIGS. 8A–C are a flowchart illustrating the operation of a fast transfer protocol BLOCK READ cycle.

DETAILED DESCRIPTION

For comparison purposes and background, an illustrative prior-art file server architecture will first be described with respect to FIG. 1. FIG. 1 is an overall block diagram of a conventional prior-art Unix-based file server for Ethernet networks. It consists of a host CPU card 10 with a single microprocessor on board. The host CPU card 10 connects to an Ethernet #1 12, and it connects via a memory management unit (MMU) 11 to a large memory array 16. The host CPU card 10 also drives a keyboard, a video display, and two RS232 ports (not shown). It also connects via the MMU 11 and a standard 32-bit VME bus 20 to various peripheral devices, including an SMD disk controller 22 controlling one or two disk drives 24, a SCSI host adaptor 26 connected to a SCSI bus 28, a tape controller 30 connected to a quarter-inch tape drive 32, and possibly a network #2 controller 34 connected to a second Ethernet 36. The SMD disk controller 22 can communicate with memory array 16 by direct memory access via bus 20 and MMU 11, with either the disk controller or the MMU acting as a bus master. This configuration is illustrative; many variations are available.

The system communicates over the Ethernets using industry standard TCP/IP and NFS protocol stacks. A description of protocol stacks in general can be found in Tanenbaum, "Computer Networks" (Second Edition, Prentice Hall:, 1988). File server protocol stacks are described at pages 535–546. The Tanenbaum reference is incorporated herein by reference.

Basically, the following protocol layers are implemented in the apparatus of FIG. 1:

Network Layer

The network layer converts data packets between a formal specific to Ethernets and a format which is independent of the particular type of network used. The Ethernet-specific format which is used in the apparatus of FIG. 1 is described in Hornig, "A Standard For The Transmission of IP Datagrams Over Ethernet Networks," RFC 894 (April 1984), which is incorporated herein by reference.

The Internet Protocol (IP) Layer

This layer provides the functions necessary to deliver a package of bits (an internet datagram) from a source to a destination over an interconnected system of networks. For messages to be sent from the file server to a client, a higher level in the server calls the IP module, providing the internet address of the destination client and the message to transmit. The IP module performs any required fragmentation of the message to accommodate packet size limitations of any intervening gateway, adds internet headers to each fragment, and calls on the network layer to transmit the resulting internet datagrams. The internet header includes a local network destination address (translated from the internet address) as well as other parameters.

For messages received by the IP layer from the network layer, the IP module determines from the internet address whether the datagram is to be forwarded to another host on another network, for example on a second Ethernet such as 36 in FIG. 1, or whether it is intended for the server itself. If it is intended for another host on the second network, the IP module determines a local net address for the destination and calls on the local network layer for that network to send the datagram. If the datagram is intended for an application program within the server, the IP layer strips off the header and passes the remaining portion of the message to the appropriate next higher layer. The internet protocol standard used in the illustrative apparatus of FIG. 1 is specified in Information Sciences Institute, "Internet Protocol, DARPA Internet Program Protocol Specification," RFC 791 (September 1981), which is incorporated herein by reference.

TCP/UDP Layer

This layer is datagram service with more elaborate packaging and addressing options than the IP layer. For example, whereas an IP datagram can hold about 1,500 bytes and be addressed to hosts, UDP datagrams can hold about 64 KB and be addressed to a particular port within a host. TCP and UDP are alternative protocols at this layer; applications requiring ordered reliable delivery of streams of data may use TCP, whereas applications (such as NFS) which do not require ordered and reliable delivery may use UDP.

The prior art file server of FIG. 1 uses both TCP and UDP. It uses UDP for file server-related services, and uses TCP for certain other services which the server provides to network clients. The UDP is specified in Postel, "User Datagram Protocol," RFC 768 (Aug. 28, 1980), which is incorporated herein by reference. TCP is specified in Postel, "Transmission Control Protocol," RFC 761 (January 1980) and RFC 793 (September 1981), which is also incorporated herein by reference.

XDR/RPC Layer

This layer provides functions callable from higher level programs to run a designated procedure on a remote machine. It also provides the decoding necessary to permit a client machine to execute a procedure on the server. For example, a caller process in a client node may send a call message to the server of FIG. 1. The call message includes a specification of the desired procedure, and its parameters. The message is passed up the stack to the RPC layer, which calls the appropriate procedure within the server. When the procedure is complete, a reply message is generated and RPC passes it back down the stack and over the network to the caller client. RPC is described in Sun Microsystems, Inc., "RPC: Remote Procedure Call Protocol Specification, Version 2," RFC 1057 (June 1988), which is incorporated herein by reference.

RPC uses the XDR external data representation standard to represent information passed to and from the underlying UDP layer. XDR is merely a data encoding standard, useful for transferring data between different computer architectures. Thus, on the network side of the XDR/RPC layer, information is machine-independent; on the host application side, it may not be. XDR is described in Sun Microsystems, Inc., "XDR: External

Data Representation Standard," RFC 1014 (June 1987), which is incorporated herein by reference.

NFS Layer

The NFS ("network file system") layer is one of the programs available on the server which an RPC request can call. The combination of host address, program number, and procedure number in an RPC request can specify one remote NFS procedure to be called.

Remote procedure calls to NFS on the file server of FIG. **1** provide transparent, stateless, remote access to shared files on the disks **24**. NFS assumes a file system that is hierarchical, with directories as all but the bottom level of files. Client hosts can call any of about 20 NFS procedures including such procedures as reading a specified number of bytes from a specified file; writing a specified number of bytes to a specified file; creating, renaming and removing specified files; parsing directory trees; creating and removing directories; and reading and setting file attributes. The location on disk to which and from which data is stored and retrieved is always specified in logical terms, such as by a file handle or Inode designation and a byte offset. The details of the actual data storage are hidden from the client. The NFS procedures, together with possible higher level modules such as Unix VFS and UFS, perform all conversion of logical data addresses to physical data addresses such as drive, head, track and sector identification. NFS is specified in Sun Microsystems, Inc., "NFS: Network File System Protocol Specification," RFC 1094 (March 1989), incorporated herein by reference.

With the possible exception of the network layer, all the protocol processing described above is done in software, by a single processor in the host CPU card **10**. That is, when an Ethernet packet arrives on Ethernet **12**, the host CPU **10** performs all the protocol processing in the NFS stack, as well as the protocol processing for any other application which may be running on the host **10**. NFS procedures are run op the host CPU **10**, with access to memory **16** for both data and program code being provided via MMU **11**. Logically specified data addresses are converted to a much more physically specified form and communicated to the SMD disk controller **22** or the SCSI bus **28**, via the VME bus **20**, and all disk caching is done by the host CPU **10** through the memory **16**. The host CPU card **10** also runs procedures for performing various other functions of the file server, communicating with tape controller **30** via the VME bus **20**. Among these are client-defined remote procedures requested by client workstations.

If the server serves a second Ethernet **36**, packets from that Ethernet are transmitted to the host CPU **10** over the same VME bus **20** in the form of IP datagrams. Again, all protocol processing except for the network layer is performed by software processes running on the host CPU **10**. In addition, the protocol processing for any message that is to be sent from the server out on either of the Ethernets **12** or **36** is also done by processes running on the host CPU **10**.

It can be seen that the host CPU **10** performs an enormous amount of processing of data, especially if 5–10 clients on each of the two Ethernets are making file server requests and need to be sent responses on a frequent basis. The host CPU **10** runs a multitasking Unix operating system, so each incoming request need not wait for the previous request to be completely processed and returned before being processed. Multiple processes are activated on the host CPU **10** for performing different stages of the processing of different requests, so many requests may be in process at the same time. But there is only one CPU on the card **10**, so the processing of these requests is not accomplished in a truly parallel manner. The processes are instead merely time sliced. The CPU **10** therefore represents a major bottleneck in the processing of file server requests.

Another bottleneck occurs in MMU **11**, which must transmit both instructions and data between the CPU card **10** and the memory **16**. All data flowing between the disk drives and the network passes through this interface at least twice.

Yet another bottleneck can occur on the VME bus **20**, which must transmit data among the SMD disk controller **22**, the SCSI host adaptor **26**, the host CPU card **10**, and possibly the network #2 controller **24**.

PREFERRED EMBODIMENT-OVERALL HARDWARE ARCHITECTURE

In FIG. **2** there is shown a block diagram of a network file server **100** according to the invention. It can include multiple network controller (NC) boards, one or more file controller (FC) boards, one or more storage processor (SP) boards, multiple system memory boards, and one or more host processors. The particular embodiment shown in FIG. **2** includes four network controller boards **110***a*–**110***d*, two file controller boards **112***a*–**112***b*, two storage processors **114***a*–**114***b*, four system memory cards **116***a*–**116***d* for a total of 192 MB of memory, and one local host processor **118**. The boards **110**, **112**, **114**, **116** and **118** are connected together over a VME bus **120** on which an enhanced block transfer mode as described in the ENHANCED VMEBUS PROTOCOL application identified above may be used. Each of the four network controllers **110** shown in FIG. **2** can be connected to up to two Ethernets **122**, for a total capacity of 8 Ethernets **122***a*–**122***h*. Each of the storage processors **114** operates ten parallel SCSI busses, nine of which can each support up to three SCSI disk drives each. The tenth SCSI channel on each of the storage processors **114** is used for tape drives and other SCSI peripherals.

The host **118** is essentially a standard SunOs Unix processor, providing all the standard Sun Open Network Computing (ONC) services except NFS and IP routing. Importantly, all network requests to run a user-defined procedure are passed to the host for execution. Each of the NC boards **110**, the FC boards **112** and the SP boards **114** includes its own independent 32-bit microprocessor. These boards essentially off-load from the host processor **118** virtually all of the NFS and disk processing. Since the vast majority of messages to and from clients over the Ethernets **122** involve NFS requests and responses, the processing of these requests in parallel by the NC, FC and SP processors, with minimal involvement by the local host **118**, vastly improves file server performance. Unix is explicitly eliminated from virtually all network, file, and storage processing.

OVERALL SOFTWARE ORGANIZATION AND DATA FLOW

Prior to a detailed discussion of the hardware subsystems shown in FIG. **2**, an overview of the software structure will now be undertaken. The software organization is described in more detail in the above-identified application entitled MULTIPLE FACILITY OPERATING SYSTEM ARCHITECTURE.

Most of the elements of the software are well known in the field and are found in most networked Unix sys-

tems, but there are two components which are not: Local NFS ("LNFS") and the messaging kernel ("MK") operating system kernel. These two components will be explained first.

The Messaging Kernel

The various processors in file server **100** communicate with each other through the use of a messaging kernel running on each of the processors **110**, **112**, **114** and **118**. These processors do not share any instruction memory, so task-level communication cannot occur via straightforward procedure calls a it does in conventional Unix. Instead, the messaging kernel passes messages over VME bus **120** to accomplish all necessary inter-processor communication. Message passing is preferred over remote procedure calls for reasons of simplicity and speed.

Messages passed by the messaging kernel have a fixed 128-byte length. Within a single processor, messages are sent by reference; between processors, they are copied by the messaging kernel and then delivered to the destination process by reference. The processors of FIG. 2 have special hardware, discussed below, that can expediently exchange and buffer interprocessor messaging kernel messages.

The LNFS Local NFS interface

The 22-function NFS standard was specifically designed for stateless operation using unreliable communication. This means that neither clients nor server can be sure if they hear each other when they talk (unreliability). In practice, an in an Ethernet environment, this works well.

Within the server **100**, however, NFS level datagrams are also used for communication between processors, in particular between the network controllers **110** and the file controller **112**, and between the host processor **118** and the file controller **112**. For this internal communication to be both efficient and convenient, it is undesirable and impractical to have complete statelessness or unreliable communications. Consequently, a modified form of NFS, namely LNFS, is used for internal communication of NFS requests and responses. LNFS is used only *within* the file server **100**; the external network protocol supported by the server is precisely standard, licensed NFS. LNFS is described in more detail below.

The Network Controllers **110** each run an NFS server which, after all protocol processing is done up to the NFS layer, converts between external NFS requests and responses and internal LNFS requests and responses. For example, NFS requests arrive as RPC requests with XDR and enclosed in a UDP datagram. After protocol processing, the NFS server translates the NFS request into LNFS form and uses the messaging kernel to send the request to the file controller **112**.

The file controller runs an LNFS server which handles LNFS requests both from network controllers and from the host **118**. The LNFS server translates LNFS requests to a form appropriate for a file system server, also running on the file controller, which manages the system memory file data cache through a block I/O layer.

An overview of the software in each of the processors will now be set forth.

Network Controller 110

The optimized dataflow of the server **100** begins with the intelligent network controller **110**. This processor receives Ethernet packets from client workstations. It quickly identifies NFS-destined packets and then performs full protocol processing on them to the NFS level, passing the resulting LNFS requests directly to the file controller **112**. This protocol processing includes IP routing and reassembly, UDP demultiplexing, XDR decoding, and NFS request dispatching. The reverse steps are used to send an NFS reply back to a client. Importantly, these time-consuming activities are performed directly in the Network Controller **110**, not in the host **118**.

The server **100** uses conventional NFS ported from Sun Microsystems, Inc., Mountain View, Calif., and is NFS protocol compatible.

Non-NFS network traffic is passed directly to its destination host processor **118**.

The NCs **110** also perform their own IP routing. Each network controller **110** supports two fully parallel Ethernets. There are four network controllers in the embodiment of the server **100** shown in FIG. 2, so that server can support up to eight Ethernets. For the two Ethernets on the same network controller **110**, IP routing occurs completely within the network controller and generates no backplane traffic. Thus attaching two mutually active Ethernets to the same controller not only minimizes their inter-net transit time, but also significantly reduces backplane contention on the VME bus **120**. Routing table updates are distributed to the network controllers from the host processor **118**, which runs either the gated or routed Unix demon.

While the network controller described here is designed for Ethernet LANs, it will be understood that the invention can be used just as readily with other network types, including FDDI.

File Controller 112

In addition to dedicating a separate processor for NFS protocol processing and IP routing, the server **100** also dedicates a separate processor, the intelligent file controller **112**, to be responsible for all file system processing. It uses conventional Berkeley Unix 4.3 file system code and uses a binary-compatible data representation on disk. These two choices allow all standard file system utilities (particularly block-level tools) to run unchanged.

The file controller **112** runs the shared file system used by all NCs **110** and the host processor **118**. Both the NCs and the host processor communicate with the file controller **12** using the LNFS interface. The NCs **110** use LNFS as described above, while the host processor **118** uses LNFS as a plug-in module to SunOs's standard Virtual File System ("VFS") interface.

When an NC receives an NFS read request from a client workstation, the resulting LNFS request passes to the FC **112**. The FC **112** first searches the system memory **116** buffer cache for the requested data. If found, a reference to the buffer is returned to the NC **110**. If not found, the LRU (least recently used) cache buffer in system memory **116** is freed and reassigned for the requested block. The FC then directs the SP **114** to read the block into the cache buffer from a disk drive array. When complete. The SP so notifies the FC, which in turn notifies the NC **100**. The NC **110** then sends an NFS reply, with the data from the buffer, back to the NFS client workstation out on the network. Note that the SP **114** transfers the data into system memory **116**, if necessary, and the NC **110** transferred the data from system memory **116** to the networks. The process takes place without any involvement of the host **118**.

Storage Processor

The intelligent storage processor **114** manages all disk and tape storage operations. While autonomous, storage processors are primarily directed by the file controller **112** to move file data between system memory **116** and the disk subsystem. The exclusion of both the host **118** and the FC **112** from the actual data path helps to supply the performance needed to service many remote clients.

Additionally, coordinated by a Server Manager in the host **118**, storage processor **114** can execute server backup by moving data between the disk subsystem and tape or other archival peripherals on the SCSI channels. Further, if directly accessed by host processor **118**, SP **114** can provide a much higher performance conventional disk interface for Unix, virtual memory, and databases. In Unix nomenclature, the host processor **118** can mount boot, storage swap, and raw partitions via the storage processors **114**.

Each storage processor **114** operates ten parallel, fully synchronous SCSI channels (busses) simultaneously. Nine of these channels support three arrays of nine SCSI disk drives each, each drive in an array being assigned to a different SCSI channel. The tenth SCSI channel hosts up to seven tape and other SCSI peripherals. In addition to performing reads and writes, SP **114** performs device-level optimizations such as disk seek queue sorting, directs device error recovery, and controls DMA transfers between the devices and system memory **116**.

Host Processor **118**

The local host **118** has three main purposes: to run Unix, to provide standard ONC network services for clients, and to run a Server Manager. Since Unix and ONC are ported from the standard SunOs Release 4 and ONC Services Release 2, the server **100** can provide identically compatible high-level ONC services such as the Yellow Pages, Lock Manager, DES Key Authenticator, Auto Mounter, and Port Mapper. Sun/2 Network disk booting and more general IP internet services such as Telnet, FTP, SMTP, SNMP, and reverse ARP are also supported. Finally, print spoolers and similar Unix demons operate transparently.

The host processor **118** runs the following software modules:

TCP and Socket Layers

The Transport Control Protocol ("TCP"), which is used for certain server functions other than NFS, provides reliable bytestream communication between two processors. Socket are used to establish TCP connections.

VFS Interface

The Virtual File System ("VFS") interface is a standard SunOs file system interface. It paints a uniform file-system picture for both users and the non-file parts of the Unix operating system, hiding the details of the specific file system. Thus standard NFS, LNFS, and any local Unix file system can coexist harmoniously.

UFS Interface

The Unix File System ("UFS") interface is the traditional and well-known Unix interface for communication with local-to-the-processor disk drives. In the server **100**, it is used to occasionally mount storage processor volumes directly, without going through the file controller **112**. Normally, the host **118** uses LNFS and goes through the file controller.

Device Layer

The device layer is a standard software interface between the Unix device model and different physical device implementations. In the server **100**, disk devices are not attached to host processors directly, so the disk driver in the host's device layer uses the messaging kernel to communicate with the storage processor **114**.

Route and Port Mapper Demons

The Route and Port Mapper demons are Unix user-level background processes that maintain the Route and Port databases for packet routing. They are mostly inactive and not in any performance path.

Yellow Pages and Authentication Demon

The Yellow Pages and Authentication services are Sun-ONC standard network services. Yellow Pages is a widely used multipurpose name-to-name directory lookup service. The Authentication service uses cryptographic keys to authenticate, or validate, requests to insure that requestors have the proper privileges for any actions or data they desire.

Server Manager

The Server Manager is an administrative application suite that controls configuration, logs error and performance reports, and provides a monitoring and tuning interface for the system administrator. These functions can be exercised from either system console connected to the host **118**, or from a system administrator's workstation.

The host processor **118** is a conventional OEM Sun central processor card, Model 3E/120. It incorporates a Motorola 68020 microprocessor and 4 MB of on-board memory. Other processors, such as a SPARC-based processor, are also possible.

The structure and operation of each of the hardware components of server **100** will now be described in detail.

NETWORK CONTROLLER HARDWARE ARCHITECTURE

FIG. **3** is a block diagram showing the data path and some control paths for an illustrative one of the network controllers **110***a*. It comprises a 20 MHz 68020 microprocessor **210** connected to a 32-bit microprocessor data bus **212**. Also connected to the microprocessor data bus **212** is a 256 K byte CPU memory **214**. The low order 8 bits of the microprocessor data bus **212** are connected through a bidirectional buffer **216** to an 8-bit slow-speed data bus **218**. On the slow-speed data bus **218** is a 128 K byte EPROM **220**, a 32 byte PROM **222**, and a multi-function peripheral (MFP) **224**. The EPROM **220** contains boot code for the network controller **110***a*, while the PROM **222** stores various operating parameters such as the Ethernet addresses assigned to each of the two Ethernet interfaces on the board. Ethernet address information is read into the corresponding interface control block in the CPU memory **214** during initialization. The MFP **224** is a Motorola 68901, and performs various local functions such as timing, interrupts, and general purpose I/O. The MFP **224** also includes a UART for interfacing to an RS232 port **226**. These functions are not critical to the invention and will not be further described herein.

The low order 16 bits of the microprocessor data bus **212** are also coupled through a bidirectional buffer **230** to a 16-bit LAN data bus **232**. A LAN controller chip **234**, such as the Am7990 LANCE Ethernet controller manufactured by Advanced Micro Devices, Inc. Sunnyvale, Calif., interfaces the LAN data bus **232** with the

first Ethernet **122***a* shown in FIG. 2. Control and data for the LAN controller **234** are stored in a 512 K byte LAN memory **236**, which is also connected to the LAN data bus **232**. A specialized 16 to 32 bit FIFO chip **240**, referred to herein as a parity FIFO chip and described below, is also connected to the LAN data bus **232**. Also connected to the LAN data bus **232** is a LAN DMA controller **242**, which controls movements of packets of data between the LAN memory **236** and the FIFO chip **240**. The LAN DMA controller **242** may be a Motorola M68440 DMA controller using channel zero only.

The second Ethernet **122***b* shown in FIG. 2 connects to a second LAN data bus **252** on the network controller card **110***a* shown in FIG. 3. The LAN data bus **252** connects to the low order 16 bits of the microprocessor data bus **212** via a bidirectional buffer **250**, and has similar components to those appearing on the LAN data bus **232**. In particular, a LAN controller **254** interfaces the LAN data bus **252** with the Ethernet **122***b*, using LAN memory **256** for data and control, and a LAN DMA controller **262** controls DMA transfer of data between the LAN memory **256** and the 16-bit wide data port A of the parity FIFO **260**.

The low order 16 bits of microprocessor data bus **212** are also connected directly to another parity FIFO **270**, and also to a control port of a VME/FIFO DMA controller **272**. The FIFO **270** is used for passing messages between the CPU memory **214** and one of the remote boards **110**, **112**, **114**, **116** or **118** (FIG. 2) in a manner described below. The VME/FIFO DMA controller **272**, which supports three round-robin non-prioritized channels for copying data, controls all data transfers between one of the remote boards and any of the FIFOs **240**, **260** or **270**, as well as between the FIFOs **240** and **260**.

32-bit data bus **274**, which is connected to the 32-bit port B of each of the FIFOs **240**, **260** and **270**, is the data bus over which these transfers take place. Data bus **274** communicates with a local 32-bit bus **276** via a bidirectional pipelining latch **278**, which is also controlled by VME/FIFO DMA controller **727**, which in turn communicates with the VME bus **120** via a bidirectional buffer **280**.

The local data bus **276** is also connected to a set of control registers **282**, which are directly addressable across the VME bus **120**. The registers **282** are used mostly for system initialization and diagnostics.

The local data bus **276** is also coupled to the microprocessor data bus **212** via a bidirectional buffer **284**. When the NC **110***a* operates in slave mode, the CPU memory **214** is directly addressable from VME bus **120**. One of the remote boards can copy data directly from the CPU memory **214** via the bidirectional buffer **284**. LAN memories **236** and **256** are not directly addressed over VME bus **120**.

The parity FIFOs **240**, **260** and **270** each consist of an ASIC, the functions and operation of which are described in the Appendix. The FIFOs **240** and **260** are configured for packet data transfer and the FIFO **270** is configured for massage passing. Referring to the Appendix, the FIFOs **240** and **260** are programmed with the following bit settings in the Data Transfer Configuration Register:

Bit	Definition	Setting
0	WD Mode	N/A
1	Parity Chip	N/A
2	Parity Correct Mode	N/A
3	8/16 bits CPU & PortA interface	16 bits (1)
4	Invert Port A address 0	no (0)
5	Invert Port A address 1	yes (1)
6	Checksum Carry Wrap	yes (1)
7	Reset	no (0)

The Data Transfer Control Register is programmed as follows:

Bit	Definition	Setting
0	Enable PortA Req/Ack	yes (1)
1	Enable PortB Req/Ack	yes (1)
2	Data Transfer Direction	(as desired)
3	CPU parity enable	no (0)
4	PortA parity enable	no (0)
5	PortB parity enable	no (0)
6	Checksum Enable	yes (1)
7	PortA Master	yes (1)

Unlike the configuration used on FIFOs **240** and **260**, the microprocessor **210** is responsible for loading and unloading Port A directly. The microprocessor **210** reads an entire 32-bit word from port A with a single instruction using two port A access cycles. Port A data transfer is disabled by unsetting bits **0** (Enable PortA Req/Ack) and **7** (PortA Master) of the Data Transfer Control Register.

The remainder of the control settings in FIFO **270** are the same as those in FIFOs **240** and **260** described above.

The NC **110***a* also includes a command FIFO **290**. The command FIFO **290** includes an input port coupled to the local data bus **276**, and which is directly addressable across the VME bus **120**, and includes an output port connected to the microprocessor data bus **212**. As explained in more detail below, when one of the remote boards issues a command or response to the NC **110***a*, it does so by directly writing a 1-word (32-bit) message descriptor into NC **110***a*'s command FIFO **290**. Command FIFO **290** generates a "FIFO not empty" status to the microprocessor **210**, which then reads the message descriptor off the top of FIFO **290** and processes it. If the message is a command, then it includes a VME address at which the message is located (presumably an address in a shared memory similar to **214** on one of the remote boards). The microprocessor **210** then programs the FIFO **270** and the VME/FIFO DMA controller **272** to copy the message from the remote location into the CPU memory **214**.

Command FIFO **290** is a conventional two-port FIFO, except that additional circuitry is included for generating a Bus Error signal on VME bus **120** if an attempt is made to write to the data input port while the FIFO is full. Command FIFO **290** has space for 256 entries.

A noteworthy feature of the architecture of NC **110***a* is that the LAN buses **232** and **252** are independent of the microprocessor data bus **212**. Data packets being routed to or from an Ethernet are stored in LAN memory **236** on the LAN data bus **232** (or **256** on the LAN data bus **252**), and not in the CPU memory **214**. Data transfer between the LAN memories **236** and **256** and the Ethernets **122***a* and **122***b*, are controlled by LAN controllers **234** and **254**, respectively, while most data transfer between LAN memory **236** or **256** and a remote

port on the VME bus **120** are controlled by LAN DMA controllers **242** and **262**, FIFOs **240** and **260**, and VME/FIFO DMA controller **272**. An exception to this rule occurs when the size of the data transfer is small, e.g., less than 64 bytes, in which case microprocessor **210** copies it directly without using DMA. The microprocessor **210** is not involved in larger transfers except in initiating them and in receiving notification when they are complete.

The CPU memory **214** contains mostly instructions for microprocessor **210**, messages being transmitted to or from a remote board via FIFO **270**, and various data blocks for controlling the FIFOs, the DMA controllers and the LAN controllers. The microprocessor **210** accesses the data packets in the LAN memories **236** and **256** by directly addressing them through the bidirectional buffers **230** and **250**, respectively, for protocol processing. The local high-speed station RAM in CPU memory **214** can therefore provide zero wait state memory access for microprocessor **210** independent of network traffic. This is in sharp contrast to the prior art architecture shown in FIG. **1**, in which all data and data packets, as well as microprocessor instructions for host CPU card **10**, reside in the memory **16** and must communicate with the host CPU card **10** via the MMU **11**.

While the LAN data buses **232** and **252** are shown as separate buses in FIG. **3**, it will be understood that they may instead be implemented as a single combined bus

NETWORK CONTROLLER OPERATION

In operation, when one of the LAN controllers (such as **234**) receives a packet of information over its Ethernet **122***a*, it reads in the entire packet and stores it in corresponding LAN memory **236**. The LAN controller **234** then issues an interrupt to microprocessor **210** via MFP **224**, and the microprocessor **210** examines the status register on LAN controller **234** (via bidirectional buffer **230**) to determine that the event causing the interrupt was a "receive packet completed." In order to avoid a potential lockout of the second Ethernet **122***b* caused by the prioritized interrupt handling characteristic of MFP **224**, the microprocessor **210** does not at this time immediately process the received packet; instead, such processing is scheduled for a polling function.

When the polling function reaches the processing of the received packet, control over the packet is passed to a software link level receive module. The link level receive module then decodes the packet according to either of two different frame formats: standard Ethernet format or SNAP (IEEE 802 LCC) format. An entry in the header in the packet specifies which frame format was used. The link level driver then determines which of three types of messages is contained in the received packet: (1) IP, (2) ARP packets which can be handled by a local ARP module, or (3) ARP packets and other packet types which must be forwarded to the local host **118** (FIG. **2**) for processing. If the packet is an ARP packet which can be handled by the NC **110***a*, such as a request for the address of server **100**, then the microprocessor **210** assembles a response packet in LAN memory **236** and, in a conventional manner, causes LAN controller **234** to transmit that packet back over Ethernet **122***a*. It is noteworthy that the data manipulation for accomplishing this task is performed almost completely in LAN memory **236**, directly addressed by microprocessor **210** as controlled by instructions in CPU memory **214**. The function is accomplished also Without generating any traffic on the VME backplane **120** at all, and without disturbing the local host **118**.

If the received packet is either an ARP packet which cannot be processed completely in the NC **110***a*, or is another type of packet which requires delivery to the local host **118** (such as a client request for the server **100** to execute a client-defined procedure), then the microprocessor **210** programs LAN DMA controller **242** to load the packet from LAN memory **236** into FIFO **240**, programs FIFO **240** with the direction of data transfer, and programs DMA controller **272** to read the packet out of FIFO **240** and across the VME bus **120** into system memory **116**. In particular, the microprocessor **210** first programs the LAN DMA controller **242** with the starting address and length of the packet in LAN memory **236**, and programs the controller to begin transferring data from the LAN memory **236** to port A of parity FIFO **240** as soon as the FIFO is ready to receive data. Second, microprocessor **210** programs the VME/FIFO DMA controller **272** with the destination address in system memory **116** and the length of the data packet, and instructs the controller to begin transferring data from port B of the FIFO **260** onto VME bus **120**. Finally, the microprocessor **210** programs FIFO **240** With the direction of the transfer to take place. The transfer then proceeds entirely under the control of DMA controllers **242** and **272**, without any further involvement by microprocessor **210**.

The microprocessor **210** then sends a message to host **118** that a packet is available at a specified system memory address. The microprocessor **210** sends such a message by writing a message descriptor to a software-emulated command FIFO on the host, which copies the message from CPU memory **214** on the NC via buffer **284** and into the host's local memory, in ordinary VME block transfer mode. The host then copies the packet from system memory **116** into the host's own local memory using ordinary VME transfers.

If the packet received by NC **110***a* from the network is an IP packet, then the microprocessor **210** determines whether it is (1) an IP packet for the server **100** which is not an NFS packet; (2) an IP packet to be routed to a different network; or (3) an NFS packet. If it is an IP packet for the server **100**, but not an NFS packet, then the microprocessor **210** causes the packet to be transmitted from the LAN memory **236** to the host **118** in the same manner described above with respect to certain ARP packets.

If the IP packet is not intended for the server **100**, but rather is to be routed to a client on a different network, then the packet is copied into the LAN memory associated with the Ethernet to which the destination client is connected. If the destination client is on the Ethernet **122***b*, which is on the same NC board as the source Ethernet **122***a*, then the microprocessor **210** causes the packet to be copied from LAN memory **236** into LAN **256** and then causes LAN controller **254** to transmit it over Ethernet **122***b*. (Of course, if the two LAN data buses **232** and **252** are combined, then copying would be unnecessary; the microprocessor **210** would simply cause the LAN controller **254** to read the packet out of the same locations in LAN memory to which the packet was written by LAN controller **234**.)

The copying of a packet from LAN memory **236** to LAN memory **256** takes place similarly to the copying described above from LAN memory to system memory. For transfer sizes of 64 bytes or more, the microprocessor **210** first programs the LAN DMA controller

242 with the starting address and length of the packet in LAN memory 236, and programs the controller to begin transferring data from the LAN memory 236 into port A of parity FIFO 240 as soon as the FIFO is ready to receive data. Second, microprocessor 210 programs the LAN DMA controller 262 with a destination address in LAN memory 256 and the length of the data packet, and instructs that controller to transfer data from parity FIFO 260 into the LAN memory 256. Third, microprocessor 210 programs the VME/FIFO DMA controller 272 to clock words of data out of port B of the FIFO 240, over the data bus 274, and into port B of FIFO 260. Finally, the microprocessor 210 programs the two FIFOs 240 and 260 with the direction of the transfer to take place. The transfer then proceeds entirely under the control of DMA controllers 242, 262 and 272, without any further involvement by the microprocessor 210. Like the copying from LAN memory to system memory, if the transfer size is smaller than 64 bytes, the microprocessor 210 performs the transfer directly, without DMA.

When each of the LAN DMA controllers 242 and 262 complete their Work, they so notify microprocessor 210 by a respective interrupt provided through MFP 224. When the microprocessor 210 has received both interrupts, it programs LAN controller 254 to transmit the packet on the Ethernet 122b in a conventional manner.

Thus, IP routing between the two Ethernets in a single network controller 110 takes plaoe over data bus 274, generating no traffic over VME bus 120. Nor is the host processor 118 disturbed for such routing, in contrast to the prior art architecture of FIG. 1. Moreover, all but the shortest copying work is performed by controllers outside microprocessor 210, requiring the involvement of the microprocessor 210, and bus traffic on microprocessor data bus 212, only for the supervisory functions of programming the DMA controllers and the parity FIFOs and instructing them to begin. The VME/FIFO DMA controller 272 is programmed by loading control registers via microprocessor data bus 212; the LAN DMA controllers 242 and 262 are programmed by loading control registers on the respective controllers via the microprocessor data bus 212, respective bidirectional buffers 230 and 250, and respective LAN data buses 232 and 252, and the parity FIFOs 240 and 260 are programmed as set forth in the Appendix.

If the destination workstation of the IP packet to be routed is on an Ethernet connected to a different one of the network controllers 110, then the packet is copied into the appropriate LAN memory on the NC 110 to which that Ethernet is connected. Such copying is accomplished by first copying the packet into system memory 116, in the manner described above with respect to certain ARP packets, and then notifying the destination NC that a packet is available. When an NC is so notified, it programs its own parity FIFO and DMA controllers to copy the packet from system memory 116 into the appropriate LAN memory. It is noteworthy that though this type of IP routing does create VME bus traffic, it still does not involve the host CPU 118.

If the IP packet received over the Ethernet 122a and now stored in LAN memory 236 is an NFS packet intended for the server 100, then the microprocessor 210 performs all necessary protocol prepossessing to extract the NFS message and convert it to the local NFS (LNFS) format. This may well involve the logical concatenation of data extracted from a large number of individual IP packets stored in LAN memory 236, resulting in a linked list, in CPU memory 214, pointing to the different blocks of data in LAN memory 236 in the correct sequence.

The exact details of the LNFS format are not important for an understanding of the invention, except to note that it includes commands to maintain a directory of files which are stored on the disks attached to the storage processors 114, commands for reading and writing data to and from a file on the disks, and various configuration management and diagnostics control messages. The directory maintenance commands which are supported by LNFS include the following messages based on conventional NFS: get attributes of a file (GETATTR); set attributes of a file (SETATTR); look up a file (LOOKUP); created a file (CREATE); remove a file (REMOVE); rename a file (RENAME); created a new linked file (LINK); create a symlink (SYMLINK); remove a directory (RMDIR); and return file system statistics (STATFS). The data transfer commands supported by LNFS include read from a file (READ); write to a file (WRITE); read from a directory (READDIR); and read a link (READLINK). LNFS also supports a buffer release command (RELEASE), for notifying the file controller that an NC is finished using a specified buffer in system memory. It also supports a VOP-derived access command, for determining whether a given type access is legal for specified credential on a specified file.

If the LNFS request includes the writing of file data from the LAN memory 236 to disk, the NC 110a first requests a buffer in system memory 116 to be allocated by the appropriate FC 112 When a pointer to the buffer is returned, microprocessor 210 programs LAN DMA controller 242, parity FIFO 240 and VME/FIFO DMA controller 272 to transmit the entire block of file data to system memory 116. The only difference between this transfer and the transfer described above for transmitting IP packets and ARP packets to system memory 116 is that these data blocks will typically have portions scattered throughout LAN memory 236. The microprocessor 210 accommodates that situation by programming LAN DMA controller 242 successively for each portion of the data, in accordance with the linked list, after receiving notification that the previous portion is complete. The microprocessor 210 can program the parity FIFO 240 and the VME/FIFO DMA controller 272 once for the entire message, as long as the entire data block is to be placed contiguously in system memory 116. If it is not, then the microprocessor 210 can program the DMA controller 272 for successive blocks in the same manner LAN DMA controller 242.

If the network controller 110a receives a message from another processor in server 100, usually from file controller 112, that file data is available in system memory 116 for transmission on one of the Ethernets, for example Ethernet 122a, then the network controller 110a copies the file data into LAN memory 236 in a manner similar to the copying of file data in the opposite direction. In particular, the microprocessor 210 first programs VME/FIFO DMA controller 272 with the starting address and length of the data in system memory 116, and programs the controller to begin transferring data over the VME bus 120 into port B of parity FIFO 240 as soon as the FIFO is ready to receive data. The microprocessor 210 then programs the LAN DMA controller 242 with a destination address in LAN mem-

ory **236** and then length of the file data, and instructs that controller to transfer data from the parity FIFO **240** into the LAN memory **236**. Third, microprocessor **210** programs the parity FIFO **240** with the direction of the transfer to take place. The transfer then proceeds entirely under the control of DMA controllers **242** and **272**, without any further involvement by the microprocessor **210**. Again, if the file data is scattered in multiple blocks in system memory **116**, the microprocessor **210** programs the VME/FIFO DMA controller **272** with a linked list of the blocks to transfer in the proper order.

When each of the DMA controllers **242** and **272** complete their work, they so notify microprocessor **210** through MFP **224**. The microprocessor **210** then performs all necessary protocol processing on the LNFS message in LAN memory **236** in order to prepare the message for transmission over the Ethernet **122***a* in the form of Ethernet IP packets. As set forth above, this protocol processing is performed entirely in network controller **110***a*, without any involvement of the local host **118**.

It should be noted that the parity FIFOs are designed to move multiples of 128-byte blocks most efficiently. The data transfer size through port B is always 32-bits wide, and the VME address corresponding to the 32-bit data must be quad-byte aligned. The data transfer size for port A can be either 8 or 16 bits. For bus utilization reasons, it is set to 16 bits when the corresponding local start address is double-byte aligned, and is set at 8 bits otherwise. The TCP/IP checksum is always computed in the 16 bit mode. Therefore, the checksum word requires byte swapping if the local start address is not double-byte aligned.

Accordingly, for transfer from port B to port A of any of the FIFOs **240**, **260** or **270**, the microprocessor **210** programs the VME/FIFO DMA controller to pad the transfer count to the next 28-byte boundary. The extra 32-bit word transfers do not involve the VME bus, and only the desired number of 32-bit words will be unloaded from port A.

For transfers from port A to port B of the parity FIFO **270**, the microprocessor **210** loads port A word-by-word and forces a FIFO full indication when it is finished. The FIFO full indication enables unloading from port B. The same procedure also takes place for transfers from port A to port B of either of the parity FIFOs **240** or **260**, since transfers of fewer than 128 bytes are performed under local microprocessor control rather than under the control of LAN DMA controller **242** or **262**. For all of the FIFOs, the VME/FIFO DMA controller is programmed to unload only the desired number of 32-bit words.

FILE CONTROLLER HARDWARE ARCHITECTURE

The file controllers (FC) **112** may each be a standard off-the-shelf microprocessor board, such as one manufactured by Motorola Inc. Preferably, however, a more specialized board is used such as that shown in block diagram form in FIG. **4**.

FIG. **4** shows one of the FCs **112***a*, and it will be understood that the other FC can be identical. In many aspects it is simply a scaled-down version of the NC **110***a* shown in FIG. **3**, and in some respects it is scaled up. Like the NC **110***a*, FC **112***a* comprises a 20 MHz 68020 microprocessor **310** connected to a 32-bit microprocessor data bus **312**. Also connected to the microprocessor data bus **312** is a 256 K byte shared CPU memory **314**. The low order 8 bits of the microprocessor data bus **312** are connected through a bidirectional buffer **316** to an 8-bit slow-speed data bus **318**. On slow-speed data bus **318** are a 128 K byte PROM **320**, and a multifunction peripheral (MFP) **324**. The functions of the PROM **320** and MFP **324** are the same as those described above with respect to EPROM **220** and MFP **224** on NC **110***a*. FC **112***a* does not include PROM like the PROM **222** on NC **110***a*, but does include a parallel port **392**. The parallel port **392** is mainly for testing and diagnostics.

Like the NC **110***a*, the FC **112***a* is connected to the VME bus **120** via a bidirectional buffer **380** and a 32-bit local data bus **376**. A set of control registers **382** are connected to the local data bus **376**, and directly addressable across the VME bus **120**. The local data bus **376** is also coupled to the microprocessor data bus **312** via a bidirectional buffer **384**. This permits the direct addressability of CPU memory **314** from VME bus **120**.

FC **112***a* also includes a command FIFO **390**, which includes an input port coupled to the local data bus **376** and which is directly addressable across the VME bus **120**. The command FIFO **390** also includes an output port connected to the microprocessor data bus **312**. The structure, operation and purpose of command FIFO **390** are the same as those described above with respect to command FIFO **290** on NC **110***a*.

The FC **112***a* omits the LAN data buses **323** and **352** which are present in NC **110***a*, but instead includes a 4 megabyte 32-bit wide FC memory **396** coupled to the microprocessor data bus **312** via a bidirectional buffer **394**. As will be seen, FC memory **396** is used as a cache memory for file control information, separate from the file data information cached in system memory **116**.

The file controller embodiment shown in FIG. **4** does not include any DMA controllers, and hence cannot act as a master for transmitting or receiving data in any block transfer mode, over the VME bus **120**. Block transfers do occur with the CPU memory **314** and the FC memory **396**, however, with the FC **112***a* acting as an VME bus slave. In such transfers, the remote master addresses the CPU memory **314** or the FC memory **396** directly over the VME bus **120** through the bidirectional buffers **384** and, if appropriate, **394**.

FILE CONTROLLER OPERATION

The purpose of the FC **112***a* is basically to provide virtual file system services in response to requests provided in LNFS format by remote processors on the VME bus **120**. Most requests will come from a network controller **110**, but requests may also come from the local host **118**.

The file related commands supported by LNFS are identified above. They are all specified to the FC **112***a* in terms of logically identified disk data blocks. For example, the LNFS command for reading data from a file includes a specification of the file from which to read (file system ID (FSID) and file ID (inode)), a byte offset, and a count of the number of bytes to read. The FC **112***a* converts that identification into physical form, namely disk and sector numbers, in order to satisfy the command.

The FC **112***a* runs a conventional Fast File System (FFS or UFS), which is based on the Berkeley 4.3 VAX release. This code performs the conversion and also performs all disk data caching and control data caching. However, as previously mentioned, control data cach-

ing is performed using the FC memory **396** on FC **112***a*, whereas disk data caching is performed using the system memory **116** FIG. 2. Caching this file control information within the FC **112***a* avoids the VME bus congestion and speed degradation which would result if file control information was cached in system memory **116**.

The memory on the FC **112***a* is directly accessed over the VME bus **120** for three main purposes. First, and by far the most frequent, are accesses to FC memory **396** by an SP **114** to read or write cached file control information. These are accesses requested by FC **112***a* to write locally modified file control structures through to disk, or to read file control structures from disk. Second, the FC's CPU memory **314** is accessed directly by other processors for message transmissions from the FC **112***a* to such other processors. For example, if a data block in system memory is to be transferred to an SP **114** for writing to disk, the FC **112***a* first assembles a message in its local memory **314** requesting such a transfer. The FC **112***a* then notifies the SP **114**, which copies the message directly from the CPU memory **314** and executes the requested transfer.

A third type of direct access to the FC's local memory occurs when an LNFS client reads directory entries. When FC **112***a* receives an LNFS request to read directory entries, the FC **112***a* formats the requested directory entries in FC memory **396** and notifies the requestor of their location. The requestor then directly accesses FC memory **396** to read the entries.

The version of the UFS code on FC **112***a* includes some modifications in order to separate the two caches. In particular, two sets of buffer headers are maintained, one for the FC memory **396** and one for the system memory **116**. Additionally, a second set of the system buffer routines (GETBLK(), BRELSE(), BREAD(), BWRITE(), and BREADA()) exist, one for buffer accesses to FC Mem **396** and one for buffer accesses to system memory **116**. The UFS code is further modified to call the appropriate buffer routines for FC memory **396** for accesses to file control information, and to call the appropriate buffer routines for the system memory **116** for the caching of disk data. A description of UFS may be found in chapters 2, 6, 7 and 8 of "Kernel Structure and Flow," by Rieken and Webb of .sh consulting (Santa Clara, Calif.: 1988), incorporated herein by reference.

When a read command is sent to the FC by a requestor such as a network controller, the FC first converts the file, offset and count information into disk and sector information. It then locks the system memory buffers which contain that information, instructing the storage processor **114** to read them from disk if necessary. When the buffer is ready, the FC returns a message to the requestor containing both the attributes of the designated file and an array of buffer descriptors that identify the locations in system memory **116** holding the data.

After the requestor has read the data out of the buffers, it sends a release request back to the FC. The release request is the same message that was returned by the FC in response to the read request; the FC **112***a* uses the information contained therein to determine which buffers to free.

A write command is processed by FC **112***a* similarly to the read command, but the caller is expected to write to (instead of read from) the locations in system memory **116** identified by the buffer descriptors returned by the FC **112***a*. Since FC **112***a* employs write-through caching, when it receives the release command from the requestor, it instructs storage processor **114** to copy the data from system memory **116** onto the appropriate disk sectors before freeing the system memory buffers for possible reallocation.

The READDIR transaction is similar to read and write, but the request is satisfied by the FC **112***a* directly out of its own FC memory **396** after formatting the requested directory information specifically for this purpose. The FC **112***a* causes the storage processor read the requested directory information from disk if it is not already locally cached. Also, the specified offset is a "magic cookie" instead of a byte offset, identifying directory entries instead of an absolute byte offset into the file. No file attributes are returned.

The READLINK transaction also returns no file attributes, and since links are always read in their entirety, it does not require any offset or count.

For all of the disk data caching performed through system memory **116**, the FC **112***a* acts as a central authority for dynamically allocating, deallocating and keeping track of buffers. If there are two or more FCs **112**, each has exclusive control over its own assigned portion of system memory **116**. In all of these transactions, the requested buffers are locked during the period between the initial request and the release request. This prevents corruption of the data by other clients.

Also in the situation where there are two or more FCs, each file system on the disks is assigned to a particular one of the FCs. FC #0 runs a process called FC_VICE_PRESIDENT, which maintains a list of which file systems are assigned to which FC. When a client processor (for example an NC **110**) is about to make an LNFS request designating a particular file system, it first sends the fsid in a message to the FC_VICE_PRESIDENT asking which FC controls the specified file system. The FC_VICE_PRESIDENT responds, and the client processor sends the LNFS request to the designated FC. The client processor also maintains its own list of fsid/FC pairs as it discovers them, so as to minimize the number of such requests to the FC_VICE_PRESIDENT.

STORAGE PROCESSOR HARDWARE ARCHITECTURE

In the file server **100**, each of the storage processors **114** can interface the VME bus **120** with up to 10 different SCSI buses. Additionally, it can do so at the full usage rate of an enhanced block transfer protocol of 55 MB per second.

FIG. **5** is a block diagram of one of the SPs **114***a*. SP **114***b* is identical. SP **114***a* comprises a microprocessor **510**, which may be a Motorola 68020 microprocessor operating at 20 MHz. The microprocessor **510** is coupled over a 32-bit microprocessor data bus **512** with CPU memory **514**, which may include up to 1 MB of static RAM. The microprocessor **510** accesses instructions, data and status on its own private bus **512**, with no contention from any other source. The microprocessor **510** is the only master of bus **512**.

The low order 16 bits of the microprocessor data bus **512** interface with a control bus **516** via a bidirectional buffer **518**. The low order 8 bits of the control bus **516** interface with a slow speed bus **520** via another bidirectional buffer **522**. The slow speed bus **520** connects to an MFP **524**, similar to the MFP **224** in NC **110***a* (FIG. 3), and with a PROM **526**, similar to PROM **220** on NC

110*a*. The PROM **526** comprises 128 K bytes of EPROM which contains the functional code for SP **114***a*. Due to the width and speed of the EPROM **526**, the functional code is copied to CPU memory **514** upon reset for faster execution.

MFP **524**, like the MFP **224** on NC **110***a*, comprises a Motorola 68901 multifunction peripheral device. It provides the functions of a vectored interrupt controller, individually programmable I/O pins, four timers and a UART. The UART functions provide serial communications across an RS **232** bus (not shown in FIG. 5) for debug monitors and diagnostics. Two of the four timing functions may be used as general-purpose timers by the microprocessor **510**, either independently or in cascaded fashion. A third timer function provides the refresh clock for a DMA controller described below, and the fourth timer generates the UART clock. Additional information on the MFP **524** can be found in "MC 6890 Multi-Function Peripheral Specification," by Motorola, Inc., which is incorporated herein by reference.

The eight general-purpose I/O bits provided by MFP **524** are configured according to the following table:

Bit	Direction	Definition
7	input	Power Failure is Imminent - This functions as an early warning
6	input	SCSI Attention - A composite of the SCSI. Attentions from all 10 SCSI channels.
5	input	Channel Operation Done - A composite of the channel done bits from all 13 channels of the DMA controller, described below.
4	output	DMA Controller Enable Enables the DMA Controller to run.
3	input	VMEbus Interrupt Done - Indicates the completion of a VMEbus Interrupt.
2	input	Command Available - Indicates that the SP'S Command Fifo, described below, contains one or more command pointers.
1	output	External Interrupts Disable. Disables externally generated interrupts to the microprocessor 510.
0	output	Command Fifo Enable Enables operation of the SP'S Command Fifo. Clears the Command Fifo when reset

Commands are provided to the SP **114***a* from the VME bus **120** via a bidirectional buffer **530**, a local data bus **532**, and a command FIFO **534**. The command FIFO **534** is similar to the command FIFOs **290** and **390** on NC **110***a* and FC **112***a*, respectively, and has a depth of 256 32-bit entries. The command FIFO **534** is a write-only register as seen on the VME bus **120**, and as a read-only register as seen by microprocessor **510**. If the FIFO is full at the beginning of a write from the VME bus, a VME bus error is generated. Pointers are removed from the command FIFO **534** in the order received, and only by the microprocessor **510**. Command available status is provided through I/O bit **4** of the MFP **524**, and as a long as one or more command pointers are still within the command FIFO **534**, the command available status remains asserted.

As previously mentioned, the Sp **114***a* supports up to 10 SCSI buses or channels **540***a*-**540***j*. In the typical configuration, buses **540***a*-**540***i* support up to 3 SCSI disk drives each, and channel **540***j* supports other SCSI peripherals such as tape drives, optical disks, and so on. Physically, the SP **114***a* connects to each of the SCSI buses with an ultra-miniature D sub connector and round shielded cables. Six 50-pin cables provide 300 conductors which carry 18 signals per bus and 12 grounds. The cables attach at the front panel of the SP **114***a* and to a commutator board at the disk drive array. Standard 50-pin cables connect each SCSI device to the commutator board. Termination resistors are installed on the SP **114***a*.

The SP **114***a* supports synchronous parallel data transfers up to 5 MB per second on each of the SCSI buses **540**, arbitration, and disconnect/reconnect services. Each SCSI bus **540** is connected to a respective SCSI adaptor **542**, which in the present embodiment is an AIC 6250 controller IC manufactured by Adaptec Inc., Milpitas, Calif., operating in the non-multiplexed address bus mode. The AIC 6250 is described in detail in "AIC-6250 Functional Specification," by Adaptec Inc., which is incorporated herein by reference. The SCSI adaptors **542** each provide the necessary hardware interface and low-level electrical protocol to implement its respective SCSI channel.

The 8-bit data port of each of the SCSI adaptors **542** is connected to port A of a respective one of a set of ten parity FIFOs **544***a*-**544***j*. The FIFOs **544** are the same as FIFOs **240**, **260** and **270** on NC **110***a*, and are connected and configured to provide parity covered data transfers between the 8-bit data port of the respective SCSI adaptors **542** and a 36-bit (32-bit plus 4 bits of parity) common data bus **550**. The FIFOs **544** provide handshake, status, word assembly/disassembly and speed matching FIFO buffering for this purpose. The FIFOs **544** also generate and check parity for the 32-bit bus, and for RAID 5 implementations they accumulate and check redundant data and accumulate recovered data.

All of the SCSI adaptors **542** reside at a single location of the address space of the microprocessor **510**, as do all of the parity FIFOs **544**. The microprocessor **510** selects individual controllers and FIFOs for access in pairs, by first programming a pair select register (not shown) to point to the desired pair and then reading from or writing to the control register address of the desired chip in the pair. The microprocessor **510** communicates with the control registers on the SCSI adaptors **542** via the control bus **516** and an additional bidirectional buffer **546**, and communicates with the control registers on FIFOs **544** via the control bus **516** and a bidirectional buffer **552**. Both the SCSI adaptors **542** and FIFOs **544** employ 8-bit control registers, and register addressing of the FIFOs **544** is arranged such that such registers alias in consecutive byte locations. This allows the microprocessor **510** to write to the registers as a single 32-bit register, thereby reducing instruction overhead.

The parity FIFOs **544** are each configured in their Adaptec 6250 mode. Referring to the Appendix, the FIFOs **544** are programmed with the following bit settings in the Data Transfer Configuration Register:

Bit	Definition	Setting
0	WD Mode	(0)
1	Parity Chip	(1)
2	Parity Correct Mode	(0)
3	8/16 bits CPU & PortA interface	(0)
4	Invert Port A address 0	(1)
5	Invert Port A address 1	(1)
6	Checksum Carry Wrap	(0)
7	Reset	(0)

The Data Transfer Control Register is programmed as follows:

Bit	Definition	Setting
0	Enable PortA Req/Ack	(1)
1	Enable PortB Req/Ack	(1)
2	Data Transfer Direction	as desired
3	CPU parity enable	(0)
4	PortA parity enable	(1)
5	PortB parity enable	(1)
6	Checksum Enable	(0)
7	PortA Master	(0)

In addition, bit 4 of the RAM Access Control Register (Long Burst) is programmed for 8-byte bursts.

SCSI adaptors **542** each generate a respective interrupt signal, the status of which are provided to microprocessor **510** as 10 bits of a 16-bit SCSI interrupt register **556**. The SCSI interrupt register **556** is connected to the control bus **516**. Additionally, a composite SCSI interrupt is provided through the MFP **524** whenever any one of the SCSI adaptors **542** needs servicing.

An additional parity FIFO **554** is also provided in the SP **114***a*, for message passing. Again referring to the Appendix, the parity FIFO **554** is programmed with the following bit settings in the Data Transfer Configuration Register:

Bit	Definition	Setting
0	WD Mode	(0)
1	Parity Chip	(1)
2	Parity Correct Mode	(0)
3	8/16 bits CPU & PortA interface	(1)
4	Invert Port A address 0	(1)
5	Invert Port A address 1	(1)
6	Checksum Carry Wrap	(0)
7	Reset	(0)

The Data Transfer Control Register is programmed as follows:

Bit	Definition	Setting
0	Enable PortA Req/Ack	(0)
1	Enable PortB Req/Ack	(1)
2	Data Transfer Direction	as desired
3	CPU parity enable	(0)
4	PortA parity enable	(0)
5	PortB parity enable	(1)
6	Checksum Enable	(0)
7	PortA Master	(0)

In addition, bit 4 of the RAM Access Control Register (Long Burst) is programmed for 8-byte bursts.

Port A of FIFO **554** is connected to the 16-bit control bus **516**, and port B is connected to the common data bus **550**. FIFO **554** provides one means by which the microprocessor **510** can communicate directly with the VME bus **120**, as is described in more detail below.

The microprocessor **510** manages data movement using a set of 15 channels, each of which has an unique status which indicates its current state. Channels are implemented using a channel enable register **560** and a channel status register **562**, both connected to the control bus **516**. The channel enable register **560** is a 16-bit write-only register, whereas the channel status register **562** is a 16-bit read-only register. The two registers reside at the same address to microprocessor **510**. The microprocessor **510** enables a particular channel by setting its respective bit in channel enable register **560**, and recognizes completion of the specified operation by testing for a "done" bit in the channel status register **562**. The microprocessor **510** then resets the enable bit, which causes the respective "done" bit in the channel status register **562** to be cleared.

The channels are defined as follows:

CHANNEL	FUNCTION
0-9	These channels control data movement to and from the respective FIFOs 544 via the common data bus 550. When a FIFO is enabled and a request is received from it, the channel becomes ready. Once the channel has been serviced a status of done is generated
11-10	These channels control data movement between a local data buffer 564, described below, and the VME bus 120. When enabled the channel becomes ready. Once the channel has been serviced a status of done is generated.
12	When enabled, this channel causes the DRAM in local data buffer 564 to be refreshed based on a clock which is generated by the MFP 524. The refresh consists of a burst of 16 rows. This channel does not generate a status of done.
13	The microprocessor's communication FIFO 554 is serviced by this channel. When enable is set and the FIFO 554 asserts a request then the channel becomes ready. This channel generates a status of done.
14	Low latency writes from microprocessor 510 onto the VME bus 120 are controlled by this channel. When this channel is enabled data is described below, onto the VME bus 120. This channel generates a done status.
15	This is a null channel for which neither a ready status nor done status is generated.

Channels are prioritized to allow servicing of the more critical requests first. Channel priority is assigned in a descending order starting at channel **14**. That is, in the event that all channels are requesting service, channel **14** will be the first one served.

The common data bus **550** is coupled via a bidirectional register **570** to a 36-bit junction bus **572**. A second bidirectional register **574** connects the junction bus **572** with the local data bus **532**. Local data buffer **564**, which comprises 1 MB of DRAM, with parity, is coupled bidirectionally to the junction bus **572**. It is organized to provide 256 K 32-bit words with byte parity. The SP **114***a* operates the DRAMs in page mode to support a very high data rate, which requires bursting of data instead of random single-word accesses. It will be seen that the local data buffer **564** is used to implement a RAID (redundant array of inexpensive disks) algorithm, and is not used for direct reading and writing between the VME bus **120** and a peripheral on one of the SCSI buses **540**.

A read-only register **576**, containing all zeros, is also connected to the junction bus **572**. This register is used mostly for diagnostics, initialization, and clearing of large blocks of data in system memory **116**.

The movement of data between the FIFOs **544** and **554**, the local data buffer **564**, and a remote entity such as the system memory **116** on the VME bus **120**, is all controlled by a VME/FIFO DMA controller **580**. The VME/FIFO DMA controller **580** is similar to the VME/FIFO DMA controller **272** on network controller **110***a* (FIG. **3**), and is described in the Appendix. Briefly, it includes a bit slice engine **582** and a dual-port static RAM **584**. One port of the dual-port static RAM **584** communicates over the 32-bit microprocessor data

bus **512** with microprocessor **510**, and the other port communicates over a separate 16-bit bus with the bit slice engine **582**. The microprocessor **510** places command parameters in the dual-port RAM **584**, and uses the channel enables **560** to signal the VME/FIFO DMA controller **580** to proceed with the command. The VME/FIFO DMA controller is responsible for scanning the channel status and servicing requests, and returning ending status in the dual-port RAM **584**. The dual-port RAM **584** is organized as 1 K×32 bits at the 32-bit port and as 2 K×16 bits at the 16-bit port. An example showing the method by which the microprocessor **510** controls the VME/FIFO DMA controller **580** is as follows. First, the microprocessor **510** writes into the dual-port RAM **584** the desired command and associated parameters for the desired channel. For example, the command might be, "copy a block of data from FIFO **544***h* out into a block of system memory **116** beginning at a specified VME address." Second, the microprocessor sets the channel enable bit in channel enable register **560** for the desired channel.

At the time the channel enable bit is set, the appropriate FIFO may not yet be ready to send data. Only when the VME/FIFO DMA controller **580** does receive a "ready" status from the channel, will the controller **580** execute the command. In the meantime, the DMA controller **580** is free to execute commands and move data to or from other channels.

When the DMA controller **580** does receive a status of "ready" from the specified channel, the controller fetches the channel command and parameters from the dual-ported RAM **584** and executes. When the command is complete, for example all the requested data has been copied, the DMA controller writes status back into the dual-port RAM **584** and asserts "done" for the channel in channel status register **562**. The microprocessor **510** is then interrupted, at which time it reads channel status register **562** to determine which channel interrupted. The microprocessor **510** then clears the channel enable for the appropriate channel and checks the ending channel status in the dual-port RAM **584**.

In this way a high-speed data transfer can take place under the control of DMA controller **580**, fully in parallel with other activities being performed by microprocessor **510**. The data transfer takes place over busses different from microprocessor data bus **512**, thereby avoiding any interference with microprocessor instruction fetches.

The SP **114***a* also includes a high-speed register **590**, which is coupled between the microprocessor data bus **512** and the local data bus **532**. The high-speed register **590** is used to write a single 32-bit word to an VME bus target with a minimum of overhead. The register is write only as viewed from the microprocessor **510**. In order to write a word onto the VME bus **120**, the microprocessor **510** first writes the word into the register **590**, and the desired VME target address into dual-port RAM **584**. When the microprocessor **510** enables the appropriate channel in channel enable register **560**, the DMA controller **580** transfers the data from the register **590** into the VME bus address specified in the dual-port RAM **584**. The DMA controller **580** then writes the ending status to the dual-port RAM and sets the channel "done" bit in channel status register **562**.

This procedure is very efficient for transfer of a single word of data, but becomes inefficient for large blocks of data. Transfers of greater than one word of data, typically for message passing, are usually performed using the FIFO **554**.

The SP **114***a* also includes a series of registers **592**, similar to the registers **282** on NC **110***a* (FIG. 3) and the registers **382** on FC **112***a* (FIG. 4). The details of these registers are not important for an understanding of the present invention.

STORAGE PROCESSOR OPERATION

The 30 SCSI disk drives supported by each of the SPs **114** are visible to a client processor, for example one of the file controllers **112**, either as three large, logical disks or as 30 independent SCSI drives, depending on configuration. When the drives are visible as three logical disks, the SP uses RAID **5** design algorithms to distribute data for each logical drive on nine physical drives to minimize disk arm contention. The tenth drive is left as a spare. The RAID **5** algorithm (redundant array of inexpensive drives, revision **5**) is described in "A Case For a Redundant Arrays of Inexpensive Disks (RAID)", by Patterson et al., published at ACM SIGMOD Conference, Chicago, Ill., Jun. 1-3, 1988, incorporated herein by reference.

In the RAID **5** design, disk data are divided into stripes. Data stripes are recorded sequentially on eight different disk drives. A ninth parity stripe, the exclusive-or of eight data stripes, is recorded on a ninth drive. If a stripe size is set to 8 K bytes, a read of 8 K of data involves only one drive. A write of 8 K of data involves two drives: a data drive and a parity drive. Since a write requires the reading back of old data to generate a new parity stripe, writes are also referred to as modify writes. The SP **114***a* supports nine small reads to nine SCSI drives concurrently. When stripe size is set to 8 K, a read of 64 K of data starts all eight SCSI drives, with each drive reading one 8 K stripe worth of data. The parallel operation is transparent to the caller client.

The parity stripes are rotated among the nine drives in order to avoid drive contention during write operations. The parity stripe is used to improve availability of data. When one drive is down, the SP **114***a* can reconstruct the missing data from a parity stripe. In such case, the SP **114***a* is running in error recovery mode. When a bad drive is repaired, the SP **114***a* can be instructed to restore data on the repaired drive while the system is on-line.

When the SP **114***a* is used to attach thirty independent SCSI drives, no parity stripe is created and the client addresses each drive directly.

The SP **114***a* processes multiple messages (transactions, commands) at one time, up to 200 messages per second. The SP **114***a* does not initiate any messages after initial system configuration. The following SP **114***a* operations are defined:

```
01 No Op
02 Send Configuration Data
03 Receive Configuration Data
05 Read and Write Sectors
06 Read and Write Cache Pages
07 IOCTL Operation
08 Dump SP 114a Local Data Buffer
09 Start/Stop A SCSI Drive
0C Inquiry
0E Read Message Log Buffer
0F Set SP 114a Interrupt
```

The above transactions are described in detail in the above-identified application entitled MULTIPLE FA-

CILITY OPERATING SYSTEM ARCHITECTURE. For and understanding of the invention, it will be useful to describe the function and operation of only two of these commands: read and write sectors, and read and write cache pages.

Read and Write Sectors

This command, issued usually by an FC **112**, causes the SP **114***a* to transfer data between a specified block of system memory and a specified series of contiguous sectors on the SCSI disks. As previously described in connection with the file controller **112**, the particular sectors are identified in physical terms. In particular, the particular disk sectors are identified by SCSI channel number (0–9), SCSI ID on that channel number (0–2), starting sector address on the specified drive, and a count of the number of sectors to read or write. The SCSI channel number is zero if the SP **114***a* is operating under RAID **5**.

The SP **114***a* can execute up to 30 messages on the 30 SCSI drives simultaneously. Unlike most of the commands to an SP **114**, which are processed by microprocessor **510** as soon as they appear on the command FIFO **534**, read and write sectors commands (as well as read and write cache memory commands) are first sorted and queued. Hence, they are not served in the order of arrival.

When a disk access command arrives, the microprocessor **510** determines which disk drive is targeted and inserts the message in a queue for that disk drive sorted by the target sector address. The microprocessor **510** executes commands on all the queues simultaneously, in the order present in the queue for each disk drive. In order to minimize disk arm movements, the microprocessor **510** moves back and forth among queue entries in an elevator fashion.

If no error conditions are detected from the SCSI disk drives, the command is completed normally. When a data check error condition occurs and the SP **114***a* is configured for RAID **5**, recovery actions using redundant data begin automatically. When a drive is down while the SP **114***a* is configured for RAID **5**, recovery actions similar to data check recovery take place.

Read/Write Cache Pages

This command is similar to read and write sectors, except that multiple VME addresses are provided for transferring disk data to and from system memory **116**. Each VME address points to a cache page in system memory **116**, the size of which is also specified in the command. When transferring data from a disk to system memory **116**, data are scattered to different cache pages; when writing data to a disk, data are gathered from different cache pages in system memory **116**. Hence, this operation is referred to as a scatter-gather function.

The target sectors on the SCSI disks are specified in the command in physical terms, in the same manner that they are specified for the read and write sectors command. Termination of the command with or without error conditions is the same as for the read and write sectors command.

The dual-port RAM **584** in the DMA controller **580** maintains a separate set of commands for each channel controlled by the bit slice engine **582**. As each channel completes its previous operation, the microprocessor **510** writes a new DMA operation into the dual-port RAM **584** for that channel in order to satisfy the next operation on a disk elevator queue.

The commands written to the DMA controller **580** include an operation code and a code indicating whether the operation is to be performed in non-block mode, in standard VME block mode, or in enhanced block mode. The operation codes supported by DMA controller **580** are as follows:

OP CODE	OPERATION	
0	NO-OP	
1	ZEROES -> BUFFER	Move zeros from zeros register 576 to local data buffer 564.
2	ZEROES -> FIFO	Move zeros from zeros register 576 to the currently selected FIFO on common data bus 550.
3	ZEROES -> VMEbus	Move zeros from zeros register 576 out onto the VME bus 120. Used for initializing cache buffers in system memory 116.
4	VMEbus -> BUFFER	Move data from the VME bus 120 to the local data buffer 564. This operation is used during a write, to move target data intended for a down drive into the buffer for participation in redundancy generation. Used only for RAID 5 application.
5	VMEbus -> FIFO	New data to be written from VME bus onto a drive. Since RAID 5 requires redundancy data to be generated from data that is buffered in local data buffer 564, this operation will be used

-continued

OP CODE	OPERATION	
		only if the SP 114a is not configured for RAID 5.
6	VMEbus -> BUFFER & FIFO	Target data is moved from VME bus 120 to a SCSI device and is also captured in the local data buffer 564 for participation in redundancy generation. Used only if SP 114a is configured for RAID 5 operation.
7	BUFFER -> VMEbus	This operation is not used.
8	BUFFER -> FIFO	Participating data is transferred to create redundant data or recovered data on a disk drive. Used only in RAID 5 applications.
9	FIFO -> VMEbus	This operation is used to move target data directly from a disk drive onto the VME bus 120.
A	FIFO -> BUFFER	Used to move participating data for recovery and modify operations. Used only in RAID 5 applications.
B	FIFO -> VMEbus & BUFFER	This operation is used to save target data for participation in data recovery. Used only in RAID 5 applications.

SYSTEM MEMORY

FIG. **6** provides a simplified block diagram of the preferred architecture of one of the system memory cards **116***a*. Each of the other system memory cards are the same. Each memory card **116** operates as a slave on the enhanced VME bus **120** and therefore requires no on-board CPU. Rather, a timing control block **610** is sufficient to provide the necessary slave control operations. In particular, the timing control block **610**, in response to control signals from the control portion of the enhanced VME bus **120**, enables a 32-bit wide buffer **612** for an appropriate direction transfer of 32-bit data between the enhanced VME bus **120** and a multiplexer unit **614**. The multiplexer **614** provides a multiplexing and demultiplexing function, depending on data transfer direction, for a six megabit by seventy-two bit word memory array **620**. An error correction code (ECC) generation and testing unit **622** is also connected to the multiplexer **614** to generate or verify, again depending on transfer direction, eight bits of ECC data. The status of ECC verification is provided back to the timing control block **610**.

ENHANCED VME BUS PROTOCOL

VME bus **120** is physically the same as an ordinary VME bus, but each of the NCs and SPs include additional circuitry and firmware for transmitting data using an enhanced VME block transfer protocol. The enhanced protocol is described in detail in the above-identified application entitled ENHANCED VMEBUS PROTOCOL UTILIZING PSEUDOSYNCHRONOUS HANDSHAKING AND BLOCK MODE DATA TRANSFER, and summarized in the Appendix hereto. Typically transfers of SNFS file data between NCs and system memory, or between SPs and system memory, and transfers of packets being routed from one NC to another through system memory, are the only types of transfers that use the enhanced protocol in server **100**. All other data transfers on VME bus **120** use either conventional VME block transfer protocols or ordinary non-block transfer protocols.

MESSAGE PASSING

As is evident from the above description, the different processors in the server **100** communicate with each other via certain types of messages. In software, these messages are all handled by the messaging kernel, described in detail in the MULTIPLE FACILITY OPERATING SYSTEM ARCHITECTURE application cited above. In hardware, they are implemented as follows.

Each of the NCs **110**, each of the FCs **112**, and each of the SPs **114** includes a command or communication FIFO such as **290** on NC **110***a*. The host **118** also includes a command FIFO, but since the host is an unmodified purchased processor board, the FIFO is emulated in software. The write port of the command FIFO in each of the processors is directly addressable from any of the other processors over VME bus **120**.

Similarly, each of the processors except SPs **114** also includes shared memory such as CPU memory **214** on NC **110***a*. This shared memory is also directly addressable by any of the other processors in the server **100**.

If one processor, for example network controller **110***a*, is to send a message or command to a second processor, for example file controller **112***a*, then it does so as follows. First, it forms the message in its own shared memory (e.g., in CPU memory **214** on NC **110***a*). Second, the microprocessor in the sending processor directly writes a message descriptor into the command

FIFO in the receiving processor. For a command being sent from network controller **110***a* to file controller **112***a*, the microprocessor **210** would perform the write via buffer **284** on NC **110***a*, VME bus **120**, and buffer **384** on file controller **112***a*.

The command descriptor is a single 32-bit word containing in its high order 30 bits a VME address indicating the start of a quad-aligned message in the sender's shared memory. The low order two bits indicate the message type as follows:

Type	Description
0	Pointer to a new message being sent
1	Pointer to a reply message
2	Pointer to message to be forwarded
3	Pointer to message to be freed; also message acknowledgment

All messages are 128-bytes long.

When the receiving processor reaches the command descriptor on its command FIFO, it directly accesses the sender's shared memory and copies it into the receiver's own local memory. For a command issued from network controller **110***a* to file controller **112***a*, this would be an ordinary VME block or non-block mode transfer from NC CPU memory **214**, via buffer **284**, VME bus **120** and buffer **384**, into FC CPU memory **314**. The FC microprocessor **310** directly accesses NC CPU memory **214** for this purpose over the VME bus **120**.

When the receiving processor has received the command and has completed its work, it sends a reply message back to the sending processor. The reply message may be no more than the original command message unaltered, or it may be a modified version of that message or a completely new message. If the reply message is not identical to the original command message, then the receiving processor directly accesses the original sender's shared memory to modify the original command message or overwrite it completely. For replies from the FC **12***a* to the NC **110***a*, this involves an ordinary VME block or non-block mode transfer from the FC **12***a*. via buffer **384**, VME bus **120**, buffer **284** and into NC CPU memory **214**. Again, the FC microprocessor **310** directly accesses NC CPU memory **214** for this purpose over the VME bus **120**.

Whether or not the original command message has been changed, the receiving processor then writes a reply message descriptor directly into the original sender's command FIFO. The reply message descriptor contains the same VME address as the original command message descriptor, and the low order two bits of the word are modified to indicate that this is a reply message. For replies from the FC **112***a* to the NC **110***a*, the message descriptor write is accomplished by microprocessor **310** directly accessing command FIFO **290** via buffer **384**, VME bus **120** and buffer **280** on the NC. Once this is done, the receiving processor can free the buffer in its local memory containing the copy of the command message.

When the original sending processor reaches the reply message descriptor on its command FIFO, it wakes up the process that originally sent the message and permits it to continue. After examining the reply message, the original sending processor can free the original command message buffer in its own local shared memory.

As mentioned above, network controller **110***a* uses the buffer **284** data path in order to write message descriptors onto the VME bus **120**, and uses VME/FIFO DMA controller **272** together with parity FIFO **270** in order to copy messages from the VME bus **120** into CPU memory **214**. Other processors read from CPU memory **214** using the buffer **284** data path.

File controller **12***a* writes message descriptors onto the VME bus **20** using the buffer **384** data path, and copies messages from other processors' shared memory via the same data path. Both take place under the control of microprocessor **310**. Other processors copy messages from CPU memory **314** also via the buffer **384** data path.

Storage processor **114***a* writes message descriptors onto the VME bus using high-speed register **590** in the manner described above, and copies messages from other processors using DMA controller **580** and FIFO **554**. The Sp **114***a* has no shared memory, however, so it uses a buffer in system memory **116** to emulate that function. That is, before it writes a message descriptor into another processor's command FIFO, the SP **114***a* first copies the message into its own previously allocated buffer in system memory **116** using DMA controller **580** and FIFO **554**. The VME address included in the message descriptor then reflects the VME address of the message in system memory **116**.

In the host **118**, the command FIFO and shared memory are both emulated in software.

The invention has been described with respect to particular embodiments thereof, and it will be understood that numerous modifications and variations are possible within the scope of the invention.

APPENDIX A

VME/FIFO DMA Controller

In storage processor **114***a*. DMA controller **580** manages the data path under the direction of the microprocessor **510**. The DMA controller **580** is a microcoded 16-bit bit-slice implementation executing pipelined instructions at a rate of one each 62.5 ns. It is responsible for scanning the channel status **562** and servicing request with parameters stored in the dual-ported ram **584** by the microprocessor **510**. Ending status is returned in the ram **584** and interrupts are generated for the microprocessor **510**.

Control Store

The control store contains the microcoded instructions which control the DMA controller **580**. The control store consists of 6 1 K×8 proms configured to yield a 1 K×48 bit microword. Locations within the control store are addressed by the sequencer and data is presented at the input of the pipeline registers.

Sequencer

The sequencer controls program flow by generating control store addresses based upon pipeline data and various status bits. The control store address consists of 10 bits. Bits 8:0 of the control store address derive from a multiplexer having as its inputs either an ALU output or the output of an incrementer. The incrementer can be preloaded with pipeline register bits 8:0, or it can be incremented as a result of a test condition. The 1 K address range is divided into two pages by a latched flag such that the microprogram can execute from either page. Branches, however remain within the selected page. Conditional sequencing is performed by having the test condition increment the pipeline provided ad-

dress. A false condition allows execution from the pipeline address while a true condition causes execution from the address +1. The alu output is selected as an address source in order to directly vector to a routine or in order to return to a calling routine. Note that when calling a subroutine the calling routine must reside within the same page as the subroutine or the wrong page will be selected on the return.

ALU

The alu comprises a single IDT49C402A integrated circuit. It is 16 bits in width and most closely resembles four 2901s with 64 registers. The alu is used primarily for incrementing, decrementing, addition and bit manipulation. All necessary control signals originate in the control store. The IDT HIGH PERFORMANCE CMOS 1988 DATA BOOK, incorporated by reference herein, contains additional information about the alu.

Microword

The 48 bit microword comprises several fields which control various functions of the DMA controller 880. The format of the microword is defined below along with mnemonics and a description of each function.

A1<8:0>	47:39	(Alu Instruction bits 8:0) The A1 bits provide the instruction for the 49C402A alu. Refer to the IDT data book for a complete definition of the alu instructions. Note that the 19 signal input of the 49C402A is always low.
CIN	38	(Carry INput) This bit forces the carry input to the alu.
RA<5:0>	37:32	(Register A address bits 5:0) These bits select one of 64 registers as the "A" operand for the alu. These bits also provide literal bits 15:10 for the alu bus.
RB<5:0>	31:26	(Register B address bits 5:0) These bits select one of 64 registers as the "B" operand for the alu. These bits also provide literal bits 9:4 for the alu bus.
LFD	25	(Latched Flag Data) When set this bit causes the selected latched flag to be set. When reset this bit causes the selected latched flag to be cleared. This bits also functions as literal bit 3 for the alu bus.
LFS<2:0>	24:22	(Latched Flag Select bits 2:0) The meaning of these bits is dependent upon the selected source for the alu bus. In the event that the literal field is selected as the bus source then LFS<2:0> function as literal bits <2:0> otherwise the bits are used to select one of the latched flags.

LFS<2:0>	SELECTED FLAG
0	This value selects a null flag.
1	When set this bit enables the buffer clock. When reset this bit disables the buffer clock.
2	When this bit is cleared VME bus transfers, buffer operations and RAS are all disabled.
3	NOT USED
4	When set this bit enables VME bus transfers.
5	When set this bit enables buffer operations.
6	When set this bit asserts the row address strobe to the dram buffer.
7	When set this bit selects page 0 of the control store.

-continued

SRC<1:0>	20,21	(alu bus SouRCe select bits 1:0) These bits select the data source to be enabled onto the alu bus.

SRC<1:0>	Selected Source
0	alu
1	dual ported ram
2	literal
3	reserved-not defined

PF<2:0>	19:17	(Pulsed Flag select bits 2:0) These bits select a flag/signal to be pulsed

PF<2:0>	Flag
0	null
1	SGL_CLK generates a single transition of buffer clock.
2	SET_VB forces vme and buffer enable to be set
3	CL_PERR clears buffer parity error status.
4	SET_DN set channel done status for the currently selected channel.
5	INC_ADR increment dual ported ram address.
6:7	RESERVED - NOT DEFINED

DEST<3:0>	16:13	(DESTination select bits 3:0) These bits select one of 10 destinations to be loaded from the alu bus.

DEST<3:0>	Destination
0	null
1	WR_RAM causes the data on the alu bus to be written to the dual ported ram. D<15:0> → ram<15:0>
2	WR_BADD loads the data from the alu bus into the dram address counters. D<14:7> → mux addr<8:0>
3	WR_VADL loads the data from the alu bus into the least significant 2 bytes of the VME address register. D<15:2> → VME addr<15:2> D1 → ENB_ENH D0 → ENB_BLK
4	WR_VADH loads the most significant 2 bytes of the VME address register. D<15:0> → VME addr<31:16>
5	WR_RADD loads the dual ported ram address counters. D<10:0> → ram addr <10:0>
6	WR_WCNT loads the word counters. D15 → count enable* D<14:8> → count <6:0>
7	WR_CO loads the co-channel select register. D<7:4> → CO<3:0>
8	WR_NXT loads the next-channel select register. D<3:0> → NEXT<3:0>
9	WR_CUR loads the current-channel select register. D<3:0> → CURR <3:0>
10:14	RESERVED - NOT DEFINED

-continued

15	JUMP	causes the control store sequencer to select the alu data bus. D<8:0> → CS_A<8:0>
TEST<3:0>	12:9	(TEST condition select bits 3:0) Select one of 16 inputs to the test multiplexer to be used as the carry input to the incrementer.

TEST<3:0>	Condition	
0	FALSE	-always false
1	TRUE	-always true
2	ALU_COUT	-carry output of alu
3	ALU_EQ	-equals output of alu
4	ALU_OVR	-alu overflow
5	ALU_NEG	-alu negative
6	XFR_DONE	-transfer complete
7	PAR_ERR	-buffer parity error
8	TIMOUT	-bus operation timeout
9	ANY_ERR	-any error status
14:10	RESERVED	-NOT DEFINED
15	CH_RDY	-next channel ready

NEXT_A<8:0>	8:0	(NEXT Address bits 8:0) Selects an instructions from the current page of the control store for execution.

Dual Ported Ram

The dual ported ram is the medium by which command, parameters and status are communicated between the DMA controller **580** and the microprocessor **510**. The ram is organized as 1K×32 at the master port and as 2 K×16 at the DMA port. The ram may be both written and read at either port.

The ram is addressed by the DMA controller **580** by loading an 11 bit address into the address counters. Data is then read into bidirectional registers and the address counter is incremented to allow read of the next location.

Writing the ram is accomplished by loading data from the processor into the registers after loading the ram address. Successive writes may be performed on every other processor cycle.

The ram contains current block pointers, ending status, high speed bus address and parameter blocks. The following is the format of the ram:

OFFSET	31	0
0	CURR POINTER 0	STATUS 0
4	INITIAL POINTER 0	
.		
.		
58	CURR POINTER B	STATUS B
5C	INITIAL POINTER B	
60	not used	not used
64	not used	not used
68	CURR POINTER D	STATUS D
6C	INITIAL POINTER D	
70	not used	STATUS E
74	HIGH SPEED BUS ADDRESS 31:2	0 0
78	PARAMETER BLOCK 0	
.		
.		
??	PARAMETER BLOCK n	

The Initial Pointer is a 32 bit value which points the first command block of a chain. The current pointer is a sixteen bit value used by the DMA controller **580** to point to the current command block. The current command block pointer should be initialized to 0×0000 by the microprocessor **510** before enabling the channel. Upon detecting a value of 0×0000 in the current block pointer the DMA controller **580** will copy the lower 16 bits from the initial pointer to the current pointer. Once the DMA controller **580** has completed the specified operations for the parameter block the current pointer will be updated to point to the next block. In the event that no further parameter blocks are available the pointer will be set to 0×0000.

The status byte indicates the ending status for the last channel operation performed. The following status bytes are defined:

STATUS	MEANING
0	NO ERRORS
1	ILLEGAL OP CODE
2	BUS OPERATION TIMEOUT
3	BUS OPERATION ERROR
4	DATA PATH PARITY ERROR

The format of the parameter block is:

OFFSET	31	0
0	FORWARD LINK	
4	NOT USED	WORD COUNT
8	VME ADDRESS 31:2	ENH BLK
C	TERM 0 OP 0	BUF ADDR 0
.		
.		
.		
C+(4Xn)	TERM n OP n	BUF ADDR n

FORWARD LINK—The forward link points to the first word of the next parameter block for execution. It allows several parameter blocks to be initialized and chained to create a sequence of operations for execution. The forward pointer has the following format:

A31:A2 0 0

The format dictates that the parameter block must start on a quad byte boundary. A pointer of 0×00000000 is a special case which indicates no forward link exists.

WORD COUNT

The word count specifies the number of quad byte words that are to be transferred to or from each buffer address or to/from the VME address. A word count of 64 K words may be specified by initializing the word count with the value of 0. The word count has the following format:

|D15|D14|D13|D12|D11|D10|D9|D8|D7|
|D6|D5|D4|D3|D2|D1|D0|

The word count is updated by the DMA controller **580** at the completion of a transfer to/from the last specified buffer address. Word count is not updated after transferring to/from each buffer address and is therefore not an accurate indicator of the total data moved to/from the buffer. Word count represents the amount of data transferred to the VME bus or one of the FIFOs **544** or **554**.

VME ADDRESS

The VME address specifies the starting address for data transfers. Thirty bits allows the address to start at any quad byte boundary.

ENH

This bit when set selects the enhanced block transfer protocol described in the above-cited ENHANCED VMEBUS PROTOCOL UTILIZING PSEUDOSYNCHRONOUS HANDSHAKING AND BLOCK MODE DATA TRANSFER application, to be used during the VME bus transfer. Enhanced protocol will be disabled automatically when performing any transfer to or from 24 bit or 16 bit address space, when the starting address is not 8 byte aligned or when the word count is not even.

BLK

This bit when set selects the conventional VME block mode protocol to be used during the VME bus transfer. Block mode will be disabled automatically when performing any transfer to or from 16 bit address space.

BUF ADDR

The buffer address specifies the starting buffer address for the adjacent operation. Only 16 bits are available for a 1M byte buffer and as a result the starting address always falls on a 16 byte boundary. The programmer must ensure that the starting address is on a modulo 128 byte boundary. The buffer address is updated by the DMA controller **580** after completion of each data burst.

|A19|A18|A17|A16|A15|A14|A13|A12|A11|
|A10|A9|A8|A7|A6|A5|A4|

TERM

The last buffer address and operation within a parameter block is identified by the terminal bit. The DMA controller **580** continues to fetch buffer addresses and operations to perform until this bit is encountered. Once the last operation within the parameter block is executed the word counter is updated and if not equal to zero the series of operations is repeated. Once the word counter reaches zero the forward link pointer is used to access the next parameter block.

|0|0|0|0|0|0|0|T|

OP

Operations are specified by the op code. The op code byte has the following format:

|0|0|0|0|OP3|OP2|OP1|OP0|

The op codes are listed below ("FIFO" refers to any of the FIFOs **544** or **554**):

OP CODE	OPERATION
0	NO-OP
1	ZEROES -> BUFFER
2	ZEROES -> FIFO
3	ZEROES -> VMEbus
4	VMEbus -> BUFFER
5	VMEbus -> FIFO
6	VMEbus -> BUFFER & FIFO
7	BUFFER -> VMEbus
8	BUFFER -> FIFO
9	FIFO -> VMEbus
A	FIFO -> BUFFER
B	FIFO -> VMEbus & BUFFER
C	RESERVED
D	RESERVED
E	RESERVED
F	RESERVED

APPENDIX B

Enhanced VME Block Transfer Protocol

The enhanced VME block transfer protocol is a VMEbus compatible pseudo-synchronous fast transfer handshake protocol for use on a VME backplane bus having a master functional module and a slave functional module logically interconnected by a data transfer bus. The data transfer bus includes a data strobe signal line and a data transfer acknowledge signal line. To accomplish the handshake, the master transmits a data strobe signal of a given duration on the data strobe line. The master then awaits the reception of a data transfer acknowledge signal from the slave module on the data transfer acknowledge signal line. The slave then responds by transmitting data transfer acknowledge signal of a given duration on the data transfer acknowledge signal line.

Consistent with the pseudo-synchronous nature of the handshake protocol, the data to be transferred is referenced to only one signal depending upon whether the transfer operation is a READ or WRITE operation. In transferring data from the master functional unit to the slave, the master broadcasts the data to be transferred. The master asserts a data strobe signal and the slave, in response to the data strobe signal, captures the data broadcast by the master. Similarly, in transferring data from the slave to the master, the slave broadcasts the data to be transferred to the master unit. The slave then asserts a data transfer acknowledge signal and the master, in response to the data transfer acknowledge signal, captures the data broadcast by the slave.

The fast transfer protocol, while not essential to the present invention, facilitates the rapid transfer of large amounts of data across a VME backplane bus by substantially increasing the data transfer rate. These data rates are achieved by using a handshake wherein the data strobe and data transfer acknowledge signals are functionally decoupled and by specifying high current drivers for all data and control lines.

The enhanced pseudo-synchronous method of data transfer (hereinafter referred to as "fast transfer mode") is implemented so as to comply and be compatible with the IEEE VME backplane bus standard. The protocol utilizes user-defined address modifiers, defined in the VMEbus standard, to indicate use of the fast transfer mode. Conventional VMEbus functional units, capable only of implementing standard VMEbus protocols, will ignore transfers made using the fast transfer mode and, as a result, are fully compatible with functional units capable of implementing the fast transfer mode.

The fast transfer mode reduces the number of bus propagations required to accomplish a handshake from four propagations, as required under conventional VMEbus protocols, to only two bus propagations. Likewise, the number of bus propagations required to effect a BLOCK READ or BLOCK WRITE data transfer is reduced. Consequently, by reducing the propagations across the VMEbus to accomplish handshaking and data transfer functions, the transfer rate is materially increased.

The enhanced protocol is described in detail in the above-cited ENHANCED VMEBUS PROTOCOL application, and will only be summarized here. Familiarity with the conventional VME bus standards is assumed.

In the fast transfer mode handshake protocol, only two bus propagations are used to accomplish a handshake, rather than four as required by the conventional protocol. At the initiation of a data transfer cycle, the master will assert and deassert DS0* in the form of a pulse of a given duration. The deassertion of DS0* is accomplished without regard as to whether a response has been received from the slave. The master then waits for an acknowledgement from the slave. Subsequent pulsing of DS0* cannot occur until a responsive DTACK* signal is received from the slave. Upon receiving the slave's assertion of DTACK*, the master can then immediately reassert data strobe, if so desired. The fast transfer mode protocol does not require the master to wait for the deassertion of DTACK* by the slave as a condition precedent to subsequent assertions of DS0*. In the fast transfer mode, only the leading edge (i.e., the assertion) of a signal is significant. Thus, the deassertion of either DS0* or DTACK* is completely irrelevant for completion of a handshake. The fast transfer protocol does not employ the DS1* line for data strobe purposes at all.

The fast transfer mode protocol may be characterized as pseudo-synchronous as it includes both synchronous and asynchronous aspects. The fast transfer mode protocol is synchronous in character due to the fact that DSO* is asserted and deasserted without regard to a response from the slave. The asynohronous aspect of the fast transfer mode protocol is attributable to the fact that the master may not subsequently assert DS0* until a response to the prior strobe is received from the slave. Consequently, because the protocol includes both synchronous and asynchronous components, it is most accurately classified as "pseudo-synchronous."

The transfer of data during a BLOCK WRITE cycle in the fast transfer protocol is referenced only to DS0*. The master first broadcasts valid data to the slave, and then asserts DS0 to the slave. The slave is given a predetermined period of time after the assertion of DS0* in which to capture the data. Hence, slave modules must be prepared to capture data at any time, as DTACK* is not referenced during the transfer cycle.

Similarly, the transfer of data during a BLOCK READ cycle in the fast transfer protocol is referenced only to DTACK*. The master first asserts DS0*. The slave then broadcasts data to the master and then asserts DTACK*. The master is given a predetermined period of time after the assertion of DTACK in which to capture the data. Hence, master modules must be prepared to capture data at any time as DS0 is not referenced during the transfer cycle.

FIG. 7, parts A through C, is a flowchart illustrating the operations involved in accomplishing the fast transfer protocol BLOCK WRITE cycle. To initiate a BLOCK WRITE cycle, the master broadcasts the memory address of the data to be transferred and the address modifier across the DTB bus. The master also drives interrupt acknowledge signal (IACK*) high and the LWORD* signal low 701. A special address modifier, for example "1F," broadcast by the master indicates to the slave module that the fast transfer protocol will be used to accomplish the BLOCK WRITE.

The starting memory address of the data to be transferred should reside on a 64-bit boundary and the size of block of data to be transferred should be a multiple of 64 bits. In order to remain in compliance with the VMEbus standard, the block must not cross a 256 byte boundary without performing a new address cycle.

The slave modules connected to the DTB receive the address and the address modifier broadcast by the master across the bus and receive LWORD* low and IACK* high 703. Shortly after broadcasting the address and address modifier 701, the master drives the AS* signal low 705. The slave modules receive the AS* low signal 707. Each slave individually determines whether it will participate in the data transfer by determining whether the broadcasted address is valid for the slave in question 709. If the address is not valid, the data transfer does not involve that particular slave and it ignores the remainder of the data transfer cycle.

The master drives WRITE* low to indicate that the transfer cycle about to occur is a WRITE operation 711. The slave receives the WRITE* low signal 713 and, knowing that the data transfer operation is a WRITE operation, awaits receipt of a high to low transition on the DS0* signal line 715. The master will wait until both DTACK* and BERR* are high 718, which indicates that the previous slave is no longer driving the DTB.

The master proceeds to place the first segment of the data to be transferred on data lines D00 through D31, 719. After placing data on D00 through D31, the master drives DS0* low 721 and, after a predetermined interval, drives DS0* high 723.

In response to the transition of DS0* from high to low, respectively 721 and 723, the slave latches the data being transmited by the master over data lines D00 through D31, 725. The master places the next segment of the data to be transferred on data lines D00 through D31, 727, and awaits receipt of a DTACK* signal in the form of a high to low transition signal, 729 in FIG. 7B.

Referring to FIG. 7B, the slave then drives DTACK* low, 731, and, after a predetermined period of time, drives DTACK high, 733. The data latched by the slave, 725, is written to a device, which has been selected to store the data 735. The slave also increments the device address 735. The slave then waits for another transition of DS0* from high to low 737.

To commence the transfer of the next segment of the block of data to be transferred, the master drives DS0* low 739 and, after a predetermined period of time, drives DS0* high 741. In response to the transition of DS0* from high to low, respectively 739 and 741, the slave latches the data being broadcast by the master over data lines D00 through D31, 743. The master places the next segment of the data to be transferred on data lines D00 through D31, 745, and awaits receipt of a DTACK* signal in the form of a high to low transition, 747.

The slave then drives DTACK* low, 749, and, after a predetermined period of time, drives DTACK* high, 751. The data latched by the slave, 743, is written to the device selected to store the data and the device address is incremented 753. The slave waits for another transition of DS0* from high to low 737.

The transfer of data will continue in the above-described manner until all of the data has been transferred from the master to the slave. After all of the data has been transferred, the master will release the address lines, address modifier lines, data lines, IACK* line, LWORD* line and DS0* line, 755. The master will then wait for receipt of a DTACK* high to low transition 757. The slave will drive DTACK* low, 759 and, after a predetermined period of time, drive DTACK* high 761. In response to the receipt of the DTACK*

high to low transition, the master will drive AS* high 763 and then release the AS* line 765.

FIG. 8, parts A through C, is a flowchart illustrating the operations involved in accomplishing the fast transfer protocol BLOCK READ cycle. To initiate a BLOCK READ cycle, the master broadcasts the memory address of the data to be transferred and the address modifier across the DTB bus 801. The master drives the LWORD* signal low and the IACK* signal high 801. As noted previously, a special address modifier indicates to the slave module that the fast transfer protocol will be used to accomplish the BLOCK READ.

The slave modules connected to the DTB receive the address and the address modifier broadcast by the master across the bus and receive LWORD* low and IACK* high 803. Shortly after broadcasting the address and address modifier 801, the master drives the AS* signal low 805. The slave modules receive the AS* low signal 807. Each slave individually determines whether it will participate in the data transfer by determining whether the broadcasted address is valid for the slave in question 809. If the address is not valid, the data transfer does not involve that particular slave and it ignores the remainder of the data transfer cycle.

The master drives WRITE* high to indicate that the transfer cycle about to occur is a READ operation 811. The slave receives the WRITE* high signal 813 and, knowing that the data transfer operation is a READ operation, places the first segment of the data to be transferred on data lines D00 through D31 819. The master will wait until both DTACK* and BERR* are high 818, which indicates that the previous slave is no longer driving the DTB.

The master then drives DS0* low 821 and, after a predetermined interval, drives DS0* high 823. The master then awaits a high to low transition on the DTACK* signal line 824. As shown in FIG. 8B, the slave then drives the DTACK* signal low 825 and after a predetermined period of time, drives the DTACK* signal high 827.

In response to the transition of DTACK* from high to low, respectively 825 and 827, the master latches the data being transmitted by the slave over data lines D00 through D31, 831. The data latched by the master, 831, is written to a device, which has been selected to store the data the device address is incremented 833.

The slave places the next segment of the data to be transferred on data lines D00 through D31, 829, and then waits for another transition of DS0* from high to low 837.

To commence the transfer of the next segment of the block of data to be transferred, the master drives DS0* low 839 and, after a predetermined period of time, drives DS0* high 841. The master then waits for the DTACK* line to transition from high to low, 843.

The slave drives DTACK* low, 845, and, after a predetermined period of time, drives DTACK* high, 847. In response to the transition of DTACK* from high to low, respectively 839 and 841, the master latches the data being transmitted by the slave over data lines D00 through D31, 845. The data latched by the master, 845, is written to the device selected to store the data, 851 in FIG. 8C, and the device address is incremented. The slave places the next segment of the data to be transferred on data lines D00 through D31, 849.

The transfer of data will continue in the above-described manner until all of the data to be transferred from the slave to the master has been written into the device selected to store the data. After all of the data to be transferred has been written into the storage device, the master will release the address lines, address modifier lines, data lines, the IACK* line, the LWORD line and DS0* line 852. The master will then wait for receipt of a DTACK* high to low transition 853. The slave will drive DTACK* low 855 and, after a predetermined period of time, drive DTACK* high 857. In response to the receipt of the DTACK* high to low transition, the master will drive AS. high B59 and release the AS* line 861.

To implement the fast transfer protocol, a conventional 64 mA tri-state driver is substituted for the 48 mA open collector driver conventionally used in VME slave modules to drive DTACK*. Similarly, the conventional VMEbus data drivers are replaced with 64 mA tri-state drivers in SO-type packages. The latter modification reduces the ground lead inductance of the actual driver package itself and, thus, reduces "ground bounce" effects which contribute to skew between data, DS0* and DTACK*. In addition, signal return inductance along the bus backplane is reduced by using a connector system having a greater number of ground pins so as to minimize signal return and mated-pair pin inductance. One such connector system is the "High Density Plus" connector, Model No. 420-8015-000, manufactured by Teradyne Corporation.

APPENDIX C

Parity FIFO

The parity FIFOs 240, 260 and 270 (on the network controllers 110), and 544 and 554 (on storage processors 114) are each implemented as an ASIC. All the parity FIFOs are identical, and are configured on power-up or during normal operation for the particular function desired. The parity FIFO is designed to allow speed matching between buses of different speed, and to perform the parity generation and correction for the parallel SCSI drives.

The FIFO comprises two bidirectional data ports, Port A and Port B, with 36×64 bits of RAM buffer between them. Port A is 8 bits wide and Port B is 32 bits wide. The RAM buffer is divided into two parts, each 36×32 bits, designated RAM X and RAM Y. The two ports access different halves of the buffer alternating to the other half when available. When the chip is configured as a parallel parity chip (e.g. one of the FIFOs 544 on SP 114a), all accesses on Port B are monitored and parity is accumulated in RAM X and RAM Y alternately.

The chip also has a CPU interface, which may be 8 or 16 bits wide. In 16 bit mode the Port A pins are used as the most significant data bits of the CPU interface and are only actually used when reading or writing to the Fifo Data Register inside the chip.

A REQ, ACK handshake is used for data transfer on both Ports A and B. The chip may be configured as either a master or a slave on Port A in the sense that, in master mode the Port A ACK / RDY output signifies that the chip is ready to transfer data on Port A, and the Port A REQ input specifies that the slave is responding. In slave mode, however, the Port A REQ input specifies that the master requires a data transfer, and the chip responds with Port A ACK / RDY when data is available. The chip is a master on Port B since it raises Port B REQ and waits for Port B ACK to indicate completion of the data transfer.

SIGNAL DESCRIPTIONS

Port A 0-7, P

Port A is the 8 bit data port. Port A P, if used, is the odd parity bit for this port.

A Req, A Ack/Rdy

These two signals are used in the data transfer mode to control the handshake of data on Port A.

uP Data 0-7, uP Data P, uPAdd 0-2, CS

These signals are used by a microprocessor to address the programmable registers within the chip. The odd parity signal uP Data P is only checked when data is written to the Fifo Data or Checksum Registers and microprocessor parity is enabled.

Clk

The clock input is used to generate some of the chip timing. It is expected to be in the 10-20 Mhz range.

Read En, Write En

During microprocessor accesses, while CS is true, these signals determine the direction of the microprocessor accesses. During data transfers in the WD mode these signals are data strobes used in conjunction with Port A Ack.

Port B 00-07, 10-17, 20-27, 30-37, 0P-3P

Port B is a 32 bit data port. There is one odd parity bit for each byte. Port B 0P is the parity of bits 00-07, PortB 1P is the parity of bits 10-17, Port B 2P is the parity of bits 20-27, and Port B 3P is the parity of bits 30-37.

B Select, B Req, B Ack, Parity Sync, B Output Enable

These signals are used in the data transfer mode to control the handshake of data on Port B. Port B Req and Port B Ack are both gated with Port B Select. The Port B Ack signal is used to strobe the data on the Port B data lines. The parity sync signal is used to indicate to a chip configured as the parity chip to indicate that the last words of data involved in the parity accumulation are on Port B. The Port B data lines will only be driven by the Fifo chip if all of the following conditions are met:

a. the data transfer is from Port A to Port B;
b. the Port B select signal is true;
c. the Port B output enable signal is true; and
d. the chip is not configured as the parity chip or it is in parity correct mode and the Parity Sync signal is true.

Reset

This signal resets all the registers within the chip and causes all bidirectional pins to be in a high impedance state.

DESCRIPTION OF OPERATION

Normal Operation

Normally the chip acts as a simple FIFO chip. A FIFO is simulated by using two RAM buffers in a simple ping-pong mode. It is intended, but not mandatory, that data is burst into or out of the FIFO on Port B. This is done by holding Port B Sel signal low and pulsing the Port B Ack signal. When transferring data from Port B to Port A, data is first written into RAM X and when this is full, the data paths will be switched such that Port B may start writing to RAM Y. Meanwhile the chip will begin emptying RAM X to Port A. When RAM Y is full and RAM X empty the data paths will be switched again such that Port B may reload RAM X and Port A may empty RAM Y.

Port A Slave Mode

This is the default mode and the chip is reset to this condition. In this mode the chip waits for a master such as one of the SCSI adapter chips 542 to raise Port A Request for data transfer. If data is available the Fifo chip will respond with Port A Ack/Rdy.

Port A WD Mode

The chip may be configured to run in the WD or Western Digital mode. In this mode the chip must be configured as a slave on Port A. It differs from the default slave mode in that the chip responds with Read Enable or Write Enable as appropriate together with Port A Ack/Rdy. This mode is intended to allow the chip to be interfaced to the Western Digital 33C93A SCSI chip or the NCR 53C90 SCSI chip.

Port A Master Mode

When the chip is configured as a master, it will raise Port A Aok/Rdy when it is ready for data transfer. This signal is expected to be tied to the Request input of a DMA controller which will respond with Port A Req when data is available. In order to allow the DMA controller to burst, the Port A Ack/Rdy signal will only be negated after every 8 or 16 bytes transferred.

Port B Parallel Write Mode

In parallel write mode, the chip is configured to be the parity chip for a parallel transfer from Port B to Port A. In this mode, when Port B Select and Port B Request are asserted, data is written into RAM X or RAM Y each time the Port B Ack signal is received. For the first block of 128 bytes data is simply copied into the selected RAM. The next 128 bytes driven on Port B will be exclusive-ORed with the first 128 bytes. This procedure will be repeated for all drives such that the parity is accumulated in this chip. The Parity Sync signal should be asserted to the parallel chip together with the last block of 128 bytes. This enables the chip to switch access to the other RAM and start accumulating a new 128 bytes of parity.

Port B Parallel Read Mode - Check Data

This mode is set if all drives are being read and parity is to be checked. In this case the Parity Correct bit in the Data Transfer Configuration Register is not set. The parity chip will first read 128 bytes on Port A as in a normal read mode and then raise Port B Request. While it has this signal asserted the chip will monitor the Port B Ack signals and exclusive-or the data on Port B with the data in its selected RAM. The Parity Sync should again be asserted with the last block of 128 bytes. In this mode the chip will not drive the Port B data lines but will check the output of its exclusive-or logic for zero. If any bits are set at this time a parallel parity error will be flagged.

Port B Parallel Read Mode - Correct Data

This mode is set by setting the Parity Correct bit in the Data Transfer Configuration Register. In this case the chip will work exactly as in the check mode except that when Port B Output Enable, Port B Select and Parity Sync are true the data is driven onto the Port B data lines and a parallel parity check for zero is not performed.

Byte Swap

In the normal mode it is expected that Port B bits 00-07 are the first byte, bits 10-17 the second byte, bits 20-27 the third byte, and bits 30-37 the last byte of each word. The order of these bytes may be changed by writing to the byte swap bits in the configuration register such that the byte address bits are inverted. The way the bytes are written and read also depend on whether the CPU interface is configured as 16 or 8 bits. The

following table shows the byte alignments for the different possibilities for data transfer using the Port A Request/Acknowledge handshake:

CPU I/F	Invert Addr 1	Invert Addr 0	Port B 00-07	Port B 10-17	Port B 20-27	Port B 30-37
8	False	False	Port A byte 0	Port A byte 1	Port A byte 2	Port A byte 1
8	False	True	Port A byte 1	Port A byte 0	Port A byte 3	Port A byte 2
8	True	False	Port A byte 2	Port A byte 3	Port A byte 0	Port A byte 1
8	True	True	Port A byte 3	Port A byte 2	Port A byte 1	Port A byte 0
16	False	False	Port A byte 0	uProc byte 0	Port A byte 1	uProc byte 1
16	False	True	uProc byte 0	Port A byte 0	uProc byte 1	Port A byte 1
16	True	False	Port A byte 1	uProc byte 1	Port A byte 0	uProc byte 0
16	True	True	uProc byte 1	Port A byte 1	uProc byte 0	Port A byte 0

When the Fifo is accessed by reading or writing the Fifo Data Register through the microprocessor port in 8 bit mode, the bytes are in the same order as the table above but the uProc data port is used instead of Port A. In 16 bit mode the table above applies.

Odd Length Transfers

If the data transfer is not a multiple of 32 words, or 128 bytes, the microprocessor must manipulate the internal registers of the chip to ensure all data is transferred. Port A Ack and Port B Req are normally not asserted until all 32 words of the selected RAM are available. These signals may be forced by writing to the appropriate RAM status bits of the Data Transfer Status Register.

When an odd length transfer has taken place the microprocessor must wait until both ports are quiescent before manipulating any registers. It should then reset both of the Enable Data Transfer bits for Port A and Port B in the Data Transfer Control Register. It must then determine by reading their Address Registers and the RAM Access Control Register whether RAM X or RAM Y holds the odd length data. It should then set the corresponding Address Register to a value of 20 hexadecimal, forcing the RAM full bit and setting the address to the first Word. Finally the microprocessor should set the Enable Data Transfer bits to allow the chip to complete the transfer.

At this point the Fifo chip will think that there are now a full 128 bytes of data in the RAM and will transfer 128 bytes if allowed to do so. The fact that some of these 128 bytes are not valid must be recognized externally to the FIFO chip.

PROGRAMMABLE REGISTERS

Data Transfer Configuration Register (Read/Write)

Register Address 0

This register is cleared by the reset signal.

Bit 0	WD Mode. Set if data transfers are to use the Western Digital WD33C93A protocol, otherwise the Adaptec 6250 protocol will be used.
Bit 1	Parity Chip. Set if this chip is to accumulate Port B parities.
Bit 2	Parity Correct Mode. Set if the parity chip is to correct parallel parity on Port B.
Bit 3	CPU Interface 16 bits wide. If set, the microprocessor data bits are combined with the Port A data bits to effectively produce a 16 bit Port. All accesses by the microprocessor as well as all data transferred using the Port A Request and Acknowledge handshake will transfer 16 bits.
Bit 4	Invert Port A byte address 0. Set to invert the least significant bit of Port A byte address.
Bit 5	Invert Port A byte address 1. Set to invert the most significant bit of Port A byte address.
Bit 6	Checksum Carry Wrap. Set to enable the carry out of the 16 bit checksum adder to carry back into the least significant bit of the adder.
Bit 7	Reset. Writing a 1 to this bit will reset the other registers. This bit resets itself after a maximum of 2 clock cycles and will therefore normally be read as a 0. No other register should be written for a minimum of 4 clock cycles after writing to this bit.

Data Transfer Control Register (Read/Write)

Register Address 1

This register is cleared by the reset signal or by writing to the reset bit.

Bit 0	Enable Data Transfer on Port A. Set to enable the Port A Req/Ack handshake.
Bit 1	Enable Data Transfer on Port B. Set to enable the Port B Req/Ack handshake.
Bit 2	Port A to Port B. If set, data transfer is from Port A to Port B. If reset, data transfer is from Port B to Port A. In order to avoid any glitches on the request lines, the state of this bit should not be altered at the same time as the enable data transfer bits 0 or 1 above.
Bit 3	uProcessor Parity Enable. Set if parity is to be checked on the microprocessor interface. It will only be checked when writing to the Fifo Data Register or reading from the Fifo Data or Checksum Registers, or during a Port A Request/Acknowledge transfer in 16 bit mode. The chip will, however, always re-generate parity ensuring that correct parity is written to the RAM or read on the microprocessor interface.
Bit 4	Port A Parity Enable. Set if parity is to be checked on Port A. It is checked when accessing the Fifo Data Register in 16 bit mode, or during a Port A Request/Acknowledge transfer. The chip will, however, always re-generate parity ensuring that correct parity is written to the RAM or read on the Port A interface.
Bit 5	Port B Parity Enable. Set if Port B data has valid byte parities. If it is not set, byte parity is generated internally to the chip when writing to the RAMs. Byte parity is not checked when writing from Port B, but always checked when reading to Port B.
Bit 6	Checksum Enable. Set to enable writing to the 16 bit checksum register. This register accumulates a 16 bit checksum for all RAM accesses, including accesses to the Fifo Data Register, as well as all writes to the checksum register. This bit must be reset before reading from the Checksum Register.

-continued

Bit 7	Port A Master. Set if Port A is to operate in the master mode on Port A during the data transfer.

Data Transfer Status Register (Read Only)

Register Address 2

This register is cleared by the reset signal or by writing to the reset bit.

Bit 0	Data in RAM X or RAM Y. Set if any bits are true in the RAM X, RAM Y, or Port A byte address registers.
Bit 1	uProc Port Parity Error. Set if the uProc Parity Enable bit is set and a parity error is detected on the microprocessor interface durng any RAM access or write to the Checksum Register in 16 bit mode.
Bit 2	Port A Parity Error. Set if the Port A Parity Enable bit is set and a parity error is detected on the Port A interface during any RAM access or write to the Checksum Register.
Bit 3	Port B Parallel Parity Error. Set if the chip is configured as the parity chip, is not in parity correct mode, and a non zero result is detected when the Parity Sync signal is true. It is also set whenever data is read out onto Port B and the data being read back through the bidirectional buffer does not compare.
Bits 4-7	Port B Bytes 0-3 Parity Error. Set whenever the data being read out of the RAMs on the Port B side has bad parity.

Ram Access Control Register (Read/Write)

Register Address 3

This register is cleared by the reset signal or by writing to the reset bit. The Enable Data Transfer bits in the Data Transfer Control Register must be reset before attempting to write to this register, else the write will be ignored.

Bit 0	Port A byte address 0. This bit is the least significant byte address bit. It is read directly bypassing any inversion done by the invert bit in the Data Transfer Configuration Register.
Bit 1	Port A byte address 1. This bit is the most significant byte address bit. It is read directly bypassing any inversion done by the invert bit in the Data Transfer Configuration Register.
Bit 2	Port A to RAM Y. Set if Port A is accessing RAM Y, and reset if it is accessing RAM X
Bit 3	Port B to RAM Y. Set if Port B is accessing RAM Y, and reset if it is accessing RAM X.
Bit 4	Long Burst. If the chip is configured to transfer data on Port A as a master, and this bit is reset, the chip will only negate Port A Ack/Rdy after every 8 bytes, or 4 words in 16 bit mode, have been transferred. If this bit is set, Port A Ack/Rdy will be negated every 16 bytes, or 8 words in 16 bit mode.
Bits 5-7	Not Used

RAM X Address Register (Read/Write)

Register Address 4

This register is cleared by the reset signal or by writing to the reset bit. The Enable Data Transfer bits in the Data Transfer Control Register must be reset before attempting to write to this register, else the write will be ignored.

Bits 0-4	RAM X word address
Bit 5	RAM X full
Bits 6-7	Not Used

RAM Y Address Register (Read/Write)

Register Address 5

This register is cleared by the reset signal or by writing to the reset bit. The Enable Data Transfer bits in the Data Transfer Control Register must be reset before attempting to write to this register, else the write will be ignored.

Bits 0-4	RAM Y word address
Bit 5	RAM Y full
Bits 6-7	Not Used

Fifo Data Register (Read/Write)

Register Address 6

The Enable Data Transfer bits in the Data Transfer Control Register must be reset before attempting to write to this register, else the write will be ignored. The Port A to Port B bit in the Data Transfer Control register must also be set before writing this register. If it is not, the RAM controls will be incremented but no data will be written to the RAM. For consistency, the Port A to PortB should be reset prior to reading this register.

Bits 0-7 are Fifo Data. The microprocessor may access the FIFO by reading or writing this register. The RAM control registers are updated as if the access was using Port A. If the chip is configured with a 16 bit CPU Interface the most significant byte will use the Port A 0-7 data lines, and each Port A access will increment the Port A byte address by 2.

Port A Checksum Register (Read/Write)

Register Address 7

This register is cleared by the reset signal or by writing to the reset bit.

Bits 0-7 are Checksum Data. The chip will accumulate a 16 bit checksum for all Port A accesses. If the chip is configured with a 16 bit CPU interface, the most significant byte is read on the Port A 0-7 data lines. If data is written directly to this register it is added to the current contents rather than overwriting them. It is important to note that the Checksum Enable bit in the Data Transfer Control Register must be set to write this register and reset to read it.

PROGRAMMING THE FIFO CHIP

In general the fifo chip is programmed by writing to the data transfer configuration and control registers to enable a data transfer, and by reading the data transfer status register at the end of the transfer to check the completion status. Usually the data transfer itself will take place with both the Port A and the Port B hand-

shakes enabled, and in this case the data transfer itself should be done without any other microprocessor interaction. In some applications, however, the Port A handshake may not be enabled, and it will be necessary for the microprocessor to fill or empty the fifo by repeatedly writing or reading the Fifo Data Register.

Since the fifo chip has no knowledge of any byte counts, there is no way of telling when any data transfer is complete by reading any register within this chip itself. Determination of whether the data transfer has been completed must therefore be done by some other circuitry outside this chip.

The following C language routines illustrate how the parity FIFO chip may be programmed. The routines assume that both Port A and the microprocessor port are connected to the system microprocessor, and return a size code of 16 bits, but that the hardware addresses the Fifo chip as long 32 bit registers.

```
struct FIFO_regs {
  unsigned char config,a1,a2,a3 ;
  unsigned char control,b1,b2,b3;
  unsigned char status,c1,c2,c3;
  unsigned char ram_access_control,d1,d2,d3;
  unsigned char ram_X_addr,e1,e2,e3;
  unsigned char ram_Y_addr,f1,f2,f3;
  unsigned long data;
  unsigned int checksum,h1;
  };

#define FIFO1 ((struct  FIFO_regs*) FIFO_BASE_ADDRESS)

#define FIFO_RESET 0x80
#define FIFO_16_BITS 0x08
#define FIFO_CARRY_WRAP 0x40
#define FIFO_PORT_A_ENABLE 0x01
#define FIFO_PORT_B_ENABLE 0x02
#define FIFO_PORT_ENABLES 0x03
#define FIFO_PORT_A_TO_B 0x04
#define FIFO_CHECKSUM_ENABLE 0x40
#define FIFO_DATA_IN_RAM 0x01
#define FIFO_FORCE_RAM_FULL 0x20

#define PORT_A_TO_PORT_B(fifo) ((fifo-> control ) & 0x04)
#define PORT_A_BYTE_ADDRESS(fifo) ((fifo->ram_access_control) &
  0x03)
#define PORT_A_TO_RAM_Y(fifo)  ((fifo->ram_access_control ) &
  0x04)
#define PORT_B_TO_RAM_Y(fifo)  ((fifo-> ram_access_control ) &
  0x08)

/*************************************************************
      The following routine initiates a Fifo data transfer using
two values passed to it.

  config_data    This is the data to be written to the
                 configuration register.

  control_data   This is the data to be written to the Data
                 Transfer Control Register.  If the data transfer
                 is to take place automatically using both the
                 Port Aand Port B handshakes, both data transfer
                 enables bits should be set in this parameter.
*************************************************************/

FIFO_initiate_data_transfer(config_data, control_data)
unsigned char config_data, control_data;
{
```

```
  FIFO1->config = config_data | FIFO_RESET;      /* Set
       Configuration value & Reset */

  FIFO1->control = control_data & (~FIFO_PORT_ENABLES);   /* Set
       everything but enables */
  FIFO1->control = control_data ;                /* Set data transfer
enables */
}

/************************************************************
       The following routine forces the transfer of any odd bytes
that have been left in the Fifo at the end of a data transfer.
It first disables both ports, then forces the Ram Full bits, and
then re-enables the appropriate Port.
************************************************************/

FIFO_force_odd_length_transfer()
{
  FIFO1->control &= ~FIFO_PORT_ENABLES; /* Disable Ports A & B */
  if (PORT_A_TO_PORT_B(FIFO1))  {
       if (PORT_A_TO_RAM_Y(FIFO1))  {
            FIFO1->ram_Y_addr = FIFO_FORCE_RAM_FULL; /* Set RAM Y
full */
       }
       else FIFO1->ram_X_addr = FIFO_FORCE_RAM_FULL ;  /* Set RAM
X full */
       FIFO1->control |= FIFO_PORT_B_ENABLE ;      /* Re-Enable
Port B */
  }
  else {
       if (PORT_B_TO_RAM_Y(FIFO1))  {
            FIFO1->ram_Y_addr = FIFO_FORCE_RAM_FULL ;        /* Set
RAM Y full */
       }
       else FIFO1->ram_X_addr = FIFO_FORCE_RAM_FULL ;  /* Set RAM
X full */
       FIFO1->control |= FIFO_PORT_A_ENABLE ;      /* Re-Enable
Port A */
  }
}

/************************************************************
       The following routine returns how many odd bytes have been
left in the Fifo at the end of a data transfer.
************************************************************/
int FIFO_count_odd_bytes()
{
  int number_odd_bytes;
  number_odd_bytes=0;
  if (FIFO1->status & FIFO_DATA_IN_RAM)   {
       if (PORT_A_TO_PORT_B(FIFO1))  {
            number_odd_bytes = (PORT_A_BYTE_ADDRESS(FIFO1)) ;
            if (PORT_A_TO_RAM_Y(FIFO1))
                 number_odd_bytes += (FIFO1->ram_Y_addr) * 4 ;
```

```
            else number_odd_bytes += (FIFO1->ram_X_addr) * 4 ;
        }
        else {
            if (PORT_B_TO_RAM_Y(FIFO1))
                number_odd_bytes = (FIFO1->ram_Y_addr) * 4 ;
            else number_odd_bytes = (FIFO1->ram_X_addr) * 4 ;
        }
    }
    return (number_odd_bytes);
}

/*************************************************************
        The following routine tests the microprocessor interface of
the chip.  It first writes and reads the first 6 registers.  It
then writes 1s, 0s, and an address pattern to the RAM, reading the
data back and checking it.

        The test returns a bit significant error code where each
bit represents the address of the registers that failed.

    Bit 0 = config register failed
    Bit 1 = control register failed
    Bit 2 = status register failed
    Bit 3 = ram access control register failed
    Bit 4 = ram X address register failed
    Bit 5 = ram Y address register failed
    Bit 6 = data register failed
    Bit 7 = checksum register failed
*************************************************************/

#define RAM_DEPTH 64        /* number of long words in Fifo Ram */

reg_expected_data[6] = { 0x7F, 0xFF, 0x00, 0x1F, 0x3F, 0x3F };

char  FIFO_uprocessor_interface_test()
{
    unsigned long test_data;
    char *register_addr;
    int i;
    char j,error;
    FIFO1->config = FIFO_RESET;         /* reset the chip */
    error=0;
    register_addr =(char *) FIFO1;
    j=1;

    /* first test registers 0 thru 5 */

    for  (i=0; i<6; i++)  {
        *register_addr = 0xFF;          /* write test data */
        if (*register_addr != reg_expected_data[i]) error |= j;
        *register_addr = 0;         /* write 0s to register */
        if (*register_addr) error |= j;
        *register_addr = 0xFF;          /* write test data again */
        if (*register_addr != reg_expected_data[i]) error |= j;
        FIFO1->config = FIFO_RESET;         /* reset the chip */
        if (*register_addr) error |= j; /* register should be 0 */
```

```
        register_addr++;          /* go to next register */
        j <<=1;
    }

        /* now test Ram data & checksum registers
        test 1s throughout Ram & then test 0s */

   for (test_data = -1; test_data != 1; test_data++)   {  /* test
 for 1s & 0s */
        FIFO1->config = FIFO_RESET | FIFO_16_BITS ;
        FIFO1->control = FIFO_PORT_A_TO_B;
        for (i=0;i<RAM_DEPTH;i++)          /* write data to RAM */
             FIFO1->data = test_data;
        FIFO1->control = 0;
        for (i=0;i<RAM_DEPTH;i++)
             if (FIFO1->data != test_data) error |= j;      /* read
 & check data */
        if (FIFO1->checksum) error |= 0x80;           /* checksum
 should = 0 */
    }

        /* now test Ram data with address pattern
         uses a different pattern for every byte */

   test_data=0x00010203;          /* address pattern start */
   FIFO1->config = FIFO_RESET | FIFO_16_BITS | FIFO_CARRY_WRAP;
   FIFO1->control = FIFO_PORT_A_TO_B | FIFO_CHECKSUM_ENABLE;
   for (i=0;i<RAM_DEPTH;i++) {
        FIFO1->data = test_data;       /* write address pattern */
        test_data += 0x04040404;
   }
   test_data=0x00010203;          /* address pattern start */
   FIFO1->control = FIFO_CHECKSUM_ENABLE;
   for (i=0;i<RAM_DEPTH;i++) {
        if (FIFO1->status != FIFO_DATA_IN_RAM)
             error |= 0x04;       /* should be data in ram */
        if (FIFO1->data != test_data) error |= j;     /* read &
 check address pattern */
        test_data += 0x04040404;
   }
   if (FIFO1->checksum != 0x0102) error |= 0x80;      /* test
 checksum of address pattern */
   FIFO1->config = FIFO_RESET | FIFO_16_BITS ;  /* inhibit carry
 wrap */
   FIFO1->checksum = 0xFEFE;      /* writing adds to checksum */
        if (FIFO1->checksum) error |=0x80;  /* checksum should be 0 */
        if (FIFO1->status) error |= 0x04;   /* status should be 0 */
        return (error);
         }
```

What is claimed is:

1. Network server apparatus for use with a data network and a mass storage device, comprising:

an interface processor unit coupleable to said network and to said mass storage device;

a host processor unit capable of running remote procedures defined by a client node on said network;

means in said interface processor unit for satisfying requests from said network to store data from said network on said mass storage device;

means in said interface processor unit for satisfying requests from said network to retrieve data from said mass storage device to said network; and

means in said interface processor unit for transmitting predefined categories of messages from said network to said host processor unit for processing in said host processor unit, said transmitted messages including all requests by a network client to run client-defined procedures on said network server apparatus.

2. Apparatus according to claim 1, wherein said interface processor unit comprises:

a network control unit coupleable to said network;

a data control unit coupleable to said mass storage device;

a buffer memory;

means in said network control unit for transmitting to said data control unit requests from said network to store specified storage data from said network on said mass storage device;

means in said network control unit for transmitting said specified storage data from said network to said buffer memory and from said buffer memory to said data control unit;

means in said network control unit for transmitting to said data control unit requests from said network to retrieve specified retrieval data from said mass storage device to said network;

means in said network control unit for transmitting said specified retrieval data from said data control unit to said buffer memory and from said buffer memory to said network; and

means in said network control unit for transmitting said predefined categories of messages from said network to said host processing unit for processing by said host processing unit.

3. Apparatus according to claim 2, wherein said data control unit comprises:

a storage processor unit coupleable to said mass storage device;

a file processor unit;

means on said file processor unit; for translating said file system level storage requests from said network into requests to store data at specified physical storage locations in said mass storage device;

means on said file processor unit for instructing said storage processor unit to write data from said buffer memory into said specified physical storage locations in said mass storage device;

means on said file processor unit for translating file system level retrieval requests from said network into requests to retrieve data from specified physical retrieval locations in said mass storage device;

means on said file processor unit for instructing said storage processor unit to retrieve data from said specified physical retrieval locations in said mass storage device to said buffer memory if said data from said specified physical locations is not already in said buffer memory; and

means in said storage processor unit for transmitting data between said buffer memory and said mass storage device.

4. Network server apparatus for use with a data network and a mass storage device, comprising:

a network control unit coupleable to said network;

a data control unit coupleable to said mass storage device;

a buffer memory;

means for transmitting from said network control unit to said data control unit requests from said network to store specified storage data from said network on said mass storage device;

means for transmitting said specified storage data by DMA from said network control unit to said buffer memory and by DMA from said buffer memory to said data control unit;

means for transmitting from said network control unit to said data control unit requests from said network to retrieve specified retrieval data from said mass storage device to said network; and

means for transmitting said specified retrieval data by DMA from said data control unit to said buffer memory and by DMA from said buffer memory to said network control unit.

5. Apparatus according to claim 1, for use further with a buffer memory, and wherein said requests from said network to store and retrieve data include file system level storage and retrieval requests respectively, and wherein said interface processor unit comprises:

a storage processor unit coupleable to said mass storage device;

a file processor unit;

means on said file processor unit for translating said file system level storage requests into requests to store data at specified physical storage locations in said mass storage device;

means on said file processor unit for instructing said storage processor unit to write data from said buffer memory into said specified physical storage locations in said mass storage device;

means on said file processor unit for translating said file system level retrieval requests into requests to retrieve data from specified physical retrieval locations in said mass storage device;

means on said file processor unit for instructing said storage processor unit to retrieve data from said specified physical retrieval locations in said mass storage device to said buffer memory if said data from said specified physical locations is not already in said buffer memory; and

means in said storage processor unit for transmitting data between said buffer memory and said mass storage device.

6. A data control unit for use with a data network and a mass storage device, and in response to file system level storage and retrieval requests from said data network, comprising:

a data bus different from said network;

a buffer memory bank coupled to said bus;

storage processor apparatus coupled to said bus and coupleable to said mass storage device;

file processor apparatus coupled to said bus, said file processor apparatus including a local memory bank

first means on said file processor unit for translating said file system level storage requests into requests to store data at specified physical storage locations in said mass storage device; and

second means on said file processor unit for translating said file system level retrieval requests into requests to retrieve data from specified physical retrieval locations in said mass storage device, said first and second means for translating collectively including means for caching file control information through said local memory bank in said file processor unit,

said data control unit further comprising means for caching the file data, to be stored or retrieved according to said storage and retrieval requests, through said buffer memory bank.

7. A network node for use with a data network and a mass storage device, comprising:

a system buffer memory;

a host processor unit having direct memory access to said system buffer memory;

a network control unit coupleable to said network and having direct memory access to said system buffer memory;

a data control unit coupleable to said mass storage device and having direct memory access to said system buffer memory;

first means for satisfying requests from said network to store data from said network on said mass storage device;

second means for satisfying requests from said network to retrieve data from said mass storage device to said network; and

third means for transmitting predefined categories of messages from said network to said host processor unit for processing in said host processor unit, said first, second and third means collectively including

means for transmitting from said network control unit to said system memory bank by direct memory access file data from said network for storage on said mass storage device,

means for transmitting from said system memory bank to said data control unit by direct memory access said file data from said network for storage on said mass storage device,

means for transmitting from said data control unit to said system memory bank by direct memory access file data for retrieval from said mass storage device to said network, and

means for transmitting from said system memory bank to said network control unit said file data for retrieval from said mass storage device to said network;

at least said network control unit including a microprocessor and local instruction storage means distinct from said system buffer memory, all instructions for said microprocessor residing in said local instruction storage means.

8. A network file server for use with a data network and a mass storage device, comprising:

a host processor unit running a Unix operating system;

an interface processor unit coupleable to said network and to said mass storage device, said interface processor unit including means for decoding all NFS requests from said network, means for performing all procedures for satisfying said NFS requests, means for encoding any NFS reply messages for return transmission on said network, and means for transmitting predefined non-NFS categories of messages from said network to said host processor unit for processing in said host processor unit.

9. Network server apparatus for use with a data network, comprising:

a network controller coupleable to said network to receive incoming information packets over said network, said incoming information packets including certain packets which contain part or all of a request to said server apparatus, said request being in either a first or a second class of requests to said server apparatus;

a first additional processor;

an interchange bus different from said network and coupled between said network controller and said first additional processor;

means in said network controller for detecting and satisfying requests in said first class of requests contained in said certain incoming information packets, said network controller lacking means in said network controller for satisfying requests in said second class of requests;

means in said network controller for detecting and assembling into assembled requests, requests in said second class of requests contained in said certain incoming information packets;

means for delivering said assembled requests from said network controller to said first additional processor over said interchange bus; and

means in said first additional processor for further processing said assembled requests in said second class of requests.

10. Apparatus according to claim 9, wherein said packets each include a network node destination address, and wherein said means in said network controller for detecting and assembling into assembled requests, assembles said assembled requests in a format which omits said network node destination addresses.

11. Apparatus according to claim 9, wherein said means in said network controller for detecting and satisfying requests in said first class of requests, assembles said requests in said first class of requests into assembled requests before satisfying said requests in said first class of requests.

12. Apparatus according to claim 9, wherein said packets each include a network node destination address, wherein said means in said network controller for detecting and assembling into assembled requests, assembles said assembled requests in a format which omits said network node destination addresses, and wherein said means in said network controller for detecting and satisfying requests in said first class of requests, assembles said requests in said first class of requests, in a format which omits said network node destination addresses, before satisfying said requests in said first class of requests.

13. Apparatus according to claim 9, wherein said means in said network controller for detecting and satisfying requests in said first class includes means for preparing an outgoing message in response to one of said first class of requests, means for packaging said outgoing message in outgoing information packets suitable for transmission over said network, and means for transmitting said outgoing information packets over said network.

14. Apparatus according to claim 9, further comprising a buffer memory coupled to said interchange bus,

and wherein said means for delivering said assembled requests comprises:

means for transferring the contents of said assembled requests over said interchange bus into said buffer memory; and

means for notifying said first additional processor of the presence of said contents in said buffer memory.

15. Apparatus according to claim 9, wherein said means in said first additional processor for further processing said assembled requests includes means for preparing an outgoing message in response to one of said second class of requests, said apparatus further comprising means for delivering said outgoing message from said first additional processor to said network controller over said interchange bus, said network controller further comprising means in said network controller for packaging said outgoing message in outgoing information packets suitable for transmission over said network, and means in said network controller for transmitting said outgoing information packages over said network.

16. Apparatus according to claim 9, wherein said first class of requests comprises requests for an address of said server apparatus, and wherein said means in said network controller for detecting and satisfying requests in said first class comprises means for preparing a response packet to such an address request and means for transmitting said response packet over said network.

17. Apparatus according to claim 9, for use further with a second data network, said network controller being coupleable further to said second network, wherein said first class of requests comprises requests to route a message to a destination reachable over said second network, and wherein said means in said network controller for detecting and satisfying requests in said first class comprises means for detecting that one of said certain packets comprises a request to route a message contained in said one of said certain packets to a destination reachable over said second network, and means for transmitting said message over said second network.

18. Apparatus according to claim 17, for use further with a third data network, said network controller further comprising means in said network controller for detecting particular requests in said incoming information packets to route a message contained in said particular requests, to a destination reachable over said third network, said apparatus further comprising:

a second network controller coupled to said interchange bus and couplable to said third data network;

means for transferring the contents of said assembled requests over said interchange bus into said buffer memory; and

means for notifying said first additional processor of the presence of said contents in said buffer memory.

means for delivering said message contained in said particular requests to said second network controller over said interchange bus; and

means in said second network controller for transmitting said message contained in said particular requests over said third network.

19. Apparatus according to claim 9, for use further with a third data network, said network controller further comprising means in said network controller for detecting particular requests in said incoming information packets to route a message contained in said particular requests, to a destination reachable over said third network, said apparatus further comprising:

a second network controller coupled to said interchange bus and couplable to said third data network;

means for delivering said message contained in said particular requests to said second network controller over said interchange bus; and

means in said second network controller for transmitting said message contained in said particular requests over said third network.

20. Apparatus according to claim 9, for use further with a mass storage device, wherein said first additional processor comprises a data control unit couplable to said mass storage device, wherein said second class of requests comprises remote calls to procedures for managing a file system in said mass storage device, and wherein said means in said first additional processor for further processing said assembled requests in said second class of requests comprises means for executing file system procedures on said mass storage device in response to said assembled requests.

21. Apparatus according to claim 20, wherein said file system procedures include a read procedure for reading data from said mass storage device,

said means in said first additional processor for further processing said assembled requests including means for reading data from a specified location in said mass storage device in response to a remote call to said read procedure,

said apparatus further including means for delivering said data to said network controller,

said network controller further comprising means on said network controller for packaging said data in outgoing information packets suitable for transmission over said network, and means for transmitting said outgoing information packets over said network.

22. Apparatus according to claim 21, wherein said means for delivering comprises:

a system buffer memory coupled to said interchange bus;

means in said data control unit for transferring said data over said interchange bus into said buffer memory; and

means in said network controller for transferring said data over said interchange bus from said system buffer memory to said network controller.

23. Apparatus according to claim 20, wherein said file system procedures include a read procedure for reading a specified number of bytes of data from said mass storage device beginning at an address specified in logical terms including a file system ID and a file ID, said means for executing file system procedures comprising:

means for converting the logical address specified in a remote call to said read procedure to a physical address; and

means for reading data from said physical address in said mass storage device.

24. Apparatus according to claim 23, wherein said mass storage device comprises a disk drive having a numbered tracks and sectors, wherein said logical address specifies said file system ID, said file ID, and a byte offset, and wherein said physical address specifies a corresponding track and sector number.

25. Apparatus according to claim 20, wherein said file system procedures include a read procedure for reading a specified number of bytes of data from said mass storage device beginning at an address specified in logical terms including a file system ID and a file ID,

said data control unit comprising a file processor coupled to said interchange bus and a storage pro-

cessor coupled to said interchange bus and coupliable to said mass storage device,

said file processor comprising means for converting the logical address specified in a remote call to said read procedure to a physical address,

said apparatus further comprising means for delivering said physical address to said storage processor,

said storage processor comprising means for reading data from said physical address in said mass storage device and for transferring said data over said interchange bus into said buffer memory; and

means in said network controller for transferring said data over said interchange bus from said system buffer memory to said network controller.

26. Apparatus according to claim **20**, wherein said file system procedures include a write procedure for writing data contained in an assembled request, to said mass storage device,

said means in said first additional processor for further processing said assembled requests including means for writing said data to a specified location in said mass storage device in response to a remote call to said read procedure.

27. Apparatus according to claim **9**, wherein said first additional processor comprises a host computer coupled to said interchange bus, wherein said second class of requests comprises remote calls to procedures other than procedures for managing a file system, and wherein said means in said first additional processor for further processing said assembled requests in said second class of requests comprises means for executing remote procedure calls in response to said assembled requests.

28. Apparatus according to claim **27**, for use further with a mass storage device and a data control unit couplable to said mass storage device and coupled to said interchange bus, wherein said network controller further comprises means in said network controller for detecting and assembling remote calls, received over said network, to procedures for managing a file system in said mass storage device, and wherein said data control unit comprises means for executing file system procedures on said mass storage device in response to said remote calls to procedures for managing a file system in said mass storage device.

29. Apparatus according to claim **27**, further comprising means for delivering all of said incoming information packets not recognized by said network controller to said host computer over said interchange bus.

30. Apparatus according to claim **9**, wherein said network controller comprises:

a microprocessor;

a local instruction memory containing local instruction code;

a local bus coupled between said microprocessor and said local instruction memory;

bus interface means for interfacing said microprocessor with said interchange bus at times determined by said microprocessor in response to said local instruction code; and

network interface means for interfacing said microprocessor with said data network,

said local instruction memory including all instruction code necessary for said microprocessor to perform said function of detecting and satisfying requests in said first class of requests, and all instruction code necessary for said microprocessor to perform said function of detecting and assembling into assembled requests, requests in said second class of requests.

31. Network server apparatus for use with a data network, comprising:

a network controller coupleable to said network to receive incoming information packets over said network, said incoming information packets including certain packets which contain part or all of a message to said server apparatus, said message being in either a first or a second class of messages to said server apparatus, said messages in said first class or messages including certain messages containing requests;

a host computer;

an interchange bus different from said network and coupled between said network controller and said host computer;

means in said network controller for detecting and satisfying said requests in said first class of messages;

means for delivering messages in said second class of messages from said network controller to said host computer over said interchange bus; and

means in said host computer for further processing said messages in said second class of messages.

32. Apparatus according to claim **31**, wherein said packets each include a network node destination address, and wherein said means for delivering messages in said second class of messages comprises means in said network controller for detecting said messages in said second class of messages and assembling them into assembled messages in a format which omits said network node destination addresses.

33. Apparatus according to claim **31**, wherein said means in said network controller for detecting and satisfying requests in said first class includes means for preparing an outgoing message in response to one of said requests in said first class of messages, means for packaging said outgoing message in outgoing information packets suitable for transmission over said network, and means for transmitting said outgoing information packets over said network.

34. Apparatus according to claim **31**, for use further with a second data network, said network controller being coupleable further to said second network, wherein said first class of messages comprises messages to be routed to a destination reachable over said second network, and wherein said means in said network controller for detecting and satisfying requests in said first class comprises means for detecting that one of said certain packets includes a request to route a message contained in said one of said certain packets to a destination reachable over said second network, and means for transmitting said message over said second network.

35. Apparatus according to claim **31**, for use further with a third data network, said network controller further comprising means in said network controller for detecting particular messages in said incoming information packets to be routed to a destination reachable over said third network, said apparatus further comprising:

a second network controller coupled to said interchange bus and couplable to said third data network;

means for delivering said particular messages to said second network controller over said interchange bus, substantially without involving said host computer; and

means in said second network controller for transmitting said message contained in said particular requests over said third network, substantially without involving said host computer.

36. Apparatus according to claim 31, for use further with a mass storage device, further comprising a data control unit coupleable to said mass storage device,

said network controller further comprising means in said network controller for detecting ones of said incoming information packets containing remote calls to procedures for managing a file system in said mass storage device, and means in said network controller for assembling said remote calls from said incoming packets into assembled calls, substantially without involving said host computer,

said apparatus further comprising means for delivering said assembled file system calls to said data control unit over said interchange bus substantially without involving said host computer,

and said data control unit comprising means in said data control unit for executing file system procedures on said mass storage device in response to said assembled file system calls, substantially without involving said host computer.

37. Apparatus according to claim 31, further comprising means for delivering all of said incoming information packets not recognized by said network controller to said host computer over said interchange bus.

38. Apparatus according to claim 31, wherein said network controller comprises:

a microprocessor;

a local instruction memory containing local instruction code;

a local bus coupled between said microprocessor and said local instruction memory;

bus interface means for interfacing said microprocessor with said interchange bus at times determined by said microprocessor in response to said local instruction code; and

network interface means for interfacing said microprocessor with said data network,

said local instruction memory including all instruction code necessary for said microprocessor to perform said function of detecting and satisfying requests in said first class of requests.

39. File server apparatus for use with a mass storage device, comprising:

a requesting unit capable of issuing calls to file system procedures in a device-independent form;

a file controller including means for converting said file system procedure calls from said device-independent form to a device-specific form and means for issuing device-specific commands in response to at least a subset of said procedure calls, said file controller operating in parallel with said requesting unit; and

a storage processor including means for executing said device-specific commands on said mass storage device, said storage processor operating in parallel with said requesting unit and said file controller.

40. Apparatus according to claim 39, further comprising:

an interchange bus;

first delivery means for delivering said file system procedure calls from said requesting unit to said file controller over said interchange bus; and

second delivery means for delivering said device-specific commands from said file controller to said storage processor over said interchange bus.

41. Apparatus according to claim 39, further comprising:

an interchange bus coupled to said requesting unit and to said file controller;

first memory means in said requesting unit and addressable over said interchange bus;

second memory means in said file controller;

means in said requesting unit for preparing in said first memory means one of said calls to file system procedures;

means for notifying said file controller of the availability of said one of said calls in said first memory means; and

means in said file controller for controlling an access to said first memory means for reading said one of said calls over said interchange bus into said second memory means in response to said notification.

42. Apparatus according to claim 41, wherein said means for notifying said file controller comprises:

a command FIFO in said file controller addressable over said interchange bus; and

means in said requesting unit for controlling an access to said FIFO for writing a descriptor into said FIFO over said interchange bus, said descriptor describing an address in said first memory means of said one of said calls and an indication that said address points to a message being sent.

43. Apparatus according to claim 41, further comprising:

means in said file controller for controlling an access to said first memory means over said interchange bus for modifying said one of said calls in said first memory means to prepare a reply to said one of said calls; and

means for notifying said requesting unit of the availability of said reply in said first memory.

44. Apparatus according to claim 41, further comprising:

a command FIFO in said requesting processor addressable over said interchange bus; and

means in said file controller for controlling an access to said FIFO for writing a descriptor into said FIFO over said interchange bus, said descriptor describing said address in said first memory and an indication that said address points to a reply to said one of said calls.

45. Apparatus according to claim 39, further comprising:

an interchange bus coupled to said file controller and to said storage processor;

second memory means in said file controller and addressable over said interchange bus;

means in said file controller for preparing one of said device-specific commands in said second memory means;

means for notifying said storage processor of the availability of said one of said commands in said second memory means; and

means in said storage processor for controlling an access to said second memory means for reading said one of said commands over said interchange bus in response to said notification.

46. Apparatus according to claim 45, wherein said means for notifying said storage processor comprises:

a command FIFO in said storage processor addressable over said interchange bus; and

means in said file controller for controlling an access to said FIFO for writing a descriptor into said FIFO over said interchange bus, said descriptor describing an address in said second memory of said one of said calls and an indication that said address points to a message being sent.

47. Apparatus according to claim 39, wherein said means for converting said file system procedure calls comprises:

a file control cache in said file controller, storing device-independent to device-specific conversion information; and

means for performing said conversions in accordance with said conversion information in said file control cache.

48. Apparatus according to claim 39, wherein said mass storage device includes a disk drive having numbered sectors, wherein one of said file system procedure calls is a read data procedure call,

said apparatus further comprising an interchange bus and a system buffer memory addressable over said interchange bus,

said means for converting said file system procedure calls including means for issuing a read sectors command in response to one of said read data procedure calls, said read sectors command specifying a starting sector on said disk drive, a count indicating the amount of data to read, and a pointer to a buffer in said system buffer memory, and

said means for executing device-specific commands including means for reading data from said disk drive beginning at said starting sector and continuing for the number of sectors indicated by said count, and controlling an access to said system buffer memory for writing said data over said interchange bus to said buffer in said system buffer memory.

49. Apparatus according to claim 48, wherein said file controller further includes means for determining whether the data specified in said one of said read data procedure calls is already present in said system buffer memory, said means for converting issuing said read sectors command only if said data is not already present in said system buffer memory.

50. Apparatus according to claim 48, further comprising:

means in said storage processor for controlling a notification of said file controller when said read sectors command has been executed;

means in said file controller, responsive to said notification from said storage processor, for controlling a notification of said requesting unit that said read data procedure call has been executed; and

means in said requesting unit, responsive to said notification from said file controller, for controlling an access to said system buffer memory for reading said data over said interchange bus from said buffer in said system buffer memory to said requesting unit.

51. Apparatus according to claim 39, wherein said mass storage device includes a disk drive having numbered sectors, wherein one of said file system procedure calls is a write data procedure call,

said apparatus further comprising an interchange bus and a system buffer memory addressable over said interchange bus,

said means for converting said file system procedure calls including means for issuing a write sectors command in response to one of said write data procedure calls, said write data procedure call including a pointer to a buffer in said system buffer memory containing data to be written, and said write sectors command including a starting sensor on said disk drive, a count indicating the amount of data to write, and said pointer to said buffer in said buffer memory, and

said means for executing device-specific commands including means for controlling an access to said buffer memory for reading said data over said interchange bus from said buffer in said system buffer memory, and writing said data to said disk drive beginning at said starting sector and continuing for the number of sectors indicated by said count.

52. Apparatus according to claim 51, further comprising:

means in said requesting unit for controlling an access to said system buffer memory for writing said data over said interchange bus to said buffer in said system buffer memory; and

means in said requesting unit for issuing said one of said write data procedure calls when said data has been written to said buffer in said system buffer memory.

53. Apparatus according to claim 52, further comprising:

means in said requesting unit for issuing a buffer allocation request; and

means in said file controller for allocating said buffer in said system buffer memory in response to said buffer allocation request, and for providing said pointer, before said data is written to said buffer in said system buffer memory.

54. Network controller apparatus for use with a first data network carrying signals representing information packets encoded according to a first physical layer protocol, comprising:

a first network interface unit, a first packet bus and first packet memory addressable by said first network interface unit over said first packet bus, said first network interface unit including means for receiving signals over said first network representing incoming information packets, extracting said incoming information packets and writing said incoming information packets into said first packet memory over said first packet bus;

a first packet bus port;

first packet DMA means for reading data over said first packet bus from said first packet memory to said first packet bus port; and

a local processor including means for accessing said incoming information packets in said first packet memory and, in response to the contents of said incoming information packets, controlling said first packet DMA means to read selected data over said first packet bus from said first packet memory to said first packet bus port, said local processor including a CPU, a CPU bus and CPU memory containing CPU instructions, said local processor operating in response to said CPU instructions, said CPU instructions being received by said CPU over said CPU bus independently of any of said writing by said first network interface unit of incoming information packets into said first packet memory over said first packet bus and independently of any

of said reading by said first packet DMA means of data over said first packet bus from said first packet memory to said first packet bus port.

55. Apparatus according to claim 54, wherein said first network interface unit further includes means for reading outgoing information packets from said first packet memory over said first packet bus, encoding said outgoing information packets according to said first physical layer protocol, and transmitting signals over said first network representing said outgoing information packets,

said local processor further including means for preparing said outgoing information packets in said first packet memory, and for controlling said first network interface unit to read, encode and transmit said outgoing information packets,

said receipt of CPU instructions by said CPU over said CPU bus being independent further of any of said reading by said first network interface unit of outgoing information packets from said first packet memory over said first packet bus.

56. Apparatus according to claim 54, further comprising a first FIFO having first and second ports, said first port of said first FIFO being said first packet bus port.

57. Apparatus according to claim 56, for use further with an interchange bus, further comprising interchange bus DMA means for reading data from said second port of said first FIFO onto said interchange bus,

said local processor further including means for controlling said interchange bus DMA means to read said data from said second port of said first FIFO onto said interchange bus.

58. Apparatus according to claim 54, for use further with a second data network carrying signals representing information packets encoded according to a second physical layer protocol, further comprising:

a second network interface unit, a second packet bus and second packet memory addressable by said second network interface unit over said second packet bus, said second network interface unit including means for reading outgoing information packets from said second packet memory over said second packet bus, encoding said outgoing information packets according to said second physical layer protocol, and transmitting signals over said second network representing said outgoing information packets;

a second packet bus port; and

second packet DMA means for reading data over said second packet bus from said second packet bus port to said second packet memory,

said local processor further including means for controlling said second packet DMA means to read data over said second packet bus from said second packet bus port to said second packet memory, and for controlling said second network interface unit to read, encode and transmit outgoing information packets from said data in said second packet memory,

said receipt of CPU instructions by said CPU over said CPU bus being independent further of any of said reading by said second packet DMA means of data over said second packet bus from said second packet bus port to said second packet memory, and independent further of any of said reading by said second network interface unit of outgoing information packets from said second packet memory over said second packet bus,

and all of said accesses to said first packet memory over said first packet bus being independent of said accesses to said second packet memory over said second packet bus.

59. Apparatus according to claim 58, wherein said second physical layer protocol is the same as said first physical layer protocol.

60. Apparatus according to claim 58, further comprising means, responsive to signals from said processor, for coupling data from said first packet bus port to said second packet bus port.

61. Apparatus according to claim 60, further comprising:

first and second FIFOs, each having first and second ports, said fist port of said first FIFO being said first packet bus port and said first port of said second FIFO being said second packet bus port;

an interchange bus; and

interchange bus DMA means for transferring data between said interchange bus and either said second port of said first FIFO or said second port of said second FIFO, selectably in response to DMA control signals from said local processor.

62. Apparatus according to claim 61, wherein said interchange bus DMA means comprises:

a transfer bus coupled to said second port of said first FIFO and to said second port of said second FIFO;

coupling means coupled between said transfer bus and said interchange bus; and

a controller coupled to receive said DMA control signals from said processor and coupled to said first and second FIFOs and to said coupling means to control data transfers over said transfer bus.

63. Storage processing apparatus for use with a plurality of storage devices on a respective plurality of channel buses, and an interchange bus, said interchange bus capable of transferring data at a higher rate than any of said channel buses, comprising:

data transfer means coupled to each of said channel buses and to said interchange bus, for transferring data in parallel between said data transfer means and each of said channel buses at the data transfer rates of each of said channel buses, respectively, and for transferring data between said data transfer means and said interchange bus at a data transfer rate higher than said data transfer rates of any of said channel buses; and

a local processor including transfer control means for controlling said data transfer means to transfer data between said data transfer means and specified ones of said channel buses and for controlling said data transfer means to transfer data between said data transfer means and said interchange bus,

said local processor including a CPU, a CPU bus and CPU memory containing CPU instructions, said local processor operating in response to said CPU instructions, said CPU instructions being received by said CPU over said CPU bus independently of any of said data transfers between said channel buses and said data transfer means and independently of any of said data transfers between said data transfer means and said interchange bus.

64. Apparatus according to claim 63, wherein the highest data transfer rate of said interchange bus is substantially equal to the sum of the highest data transfer rates of all of said channel buses.

65. Apparatus according to claim 63, wherein said data transfer means comprises:

a FIFO corresponding to each of said channel buses, each of said FIFOs having a first port and a second port;

a channel adapter coupled between the first port of each of said FIFOs and a respective one of said channels; and

DMA means coupled to the second port of each of said FIFOs and to said interchange bus, for transferring data between said interchange bus and one of said FIFOs as specified by said local processor, said transfer control means in said local processor comprising means for controlling each of said channel adapters separately to transfer data between the channel bus coupled to said channel adapter and the FIFO coupled to said channel adapter, and for controlling said DMA controller to transfer data between separately specified ones of said FIFOs and said interchange bus, said DMA means performing said transfers sequentially.

66. Apparatus according to claim 65, wherein said DMA means comprises a command memory and a DMA processor, said local processor having means for writing FIFO/interchange bus DMA commands into said command memory, each of said commands being specific to a given one said FIFOs and including an indication of the direction of data transfer between said interchange bus and said given FIFO, each of said FIFOs generating a ready status indication, said DMA processor controlling the data transfer specified in each of said commands sequentially after the corresponding FIFO indicates a ready status, and notifying said local processor upon completion of the data transfer specified in each of said commands.

67. Apparatus according to claim 65 further comprising an additional FIFO coupled between said CPU bus and said DMA memory, said local processor further having means for transferring data between said CPU and said additional FIFO, and said DMA means being further for transferring data between said interchange bus and said additional FIFO in response to commands issued by said local processor.

* * * * *

US006065037A

United States Patent [19]

Hitz et al.

[11] Patent Number: 6,065,037

[45] Date of Patent: *May 16, 2000

[54] **MULTIPLE SOFTWARE-FACILITY COMPONENT OPERATING SYSTEM FOR CO-OPERATIVE PROCESSOR CONTROL WITHIN A MULTIPROCESSOR COMPUTER SYSTEM**

[75] Inventors: **David Hitz**, Sunnyvale; **Allan Schwartz**, Saratoga; **James Lau**, Cupertino; **Guy Harris**, Mountain View, all of Calif.

[73] Assignee: **Auspex Systems, Inc.**, Santa Clara, Calif.

[*] Notice: This patent issued on a continued prosecution application filed under 37 CFR 1.53(d), and is subject to the twenty year patent term provisions of 35 U.S.C. 154(a)(2).

[21] Appl. No.: **08/473,244**

[22] Filed: **Jun. 7, 1995**

Related U.S. Application Data

[62] Division of application No. 08/225,356, Apr. 8, 1994, Pat. No. 5,485,579, which is a continuation of application No. 07/875,585, Apr. 28, 1992, abandoned, which is a continuation of application No. 07/404,885, Sep. 8, 1989, abandoned.

[51] **Int. Cl.**[7] **G06F 15/16**

[52] **U.S. Cl.** **709/200**; 709/100; 709/107; 709/300

[58] **Field of Search** 395/680, 684, 395/677, 200.2, 200.15; 709/100, 300, 107, 226, 200

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,626,634	12/1986	Brahm et al.	379/28
4,649,473	3/1987	Hammer et al.	364/200
4,709,325	11/1987	Yajima	364/200
4,903,258	2/1990	Kuhlman et al.	370/58.2
4,993,017	2/1991	Bachinger et al.	370/58.2

(List continued on next page.)

OTHER PUBLICATIONS

Kelly, Paul, "Functional Programming for Loosely-coupled Multiprocessors", The MIT Press (1989), Chaps. 1–6, pp. 1–164.

Ousterhout et al, The Sprite Network Operating System, (also published in IEEE Feb. 1988 vol. 21 issue 2), 1987.

Rogado, A Strawman Proposal for the Cluster Project, OSF Research Institute, Jul. 1992.

Draves, A Revised IPC Interface, Proceedings of the Usenix Mach Conference, Oct. 1990.

Welch, The File System Belongs in the Kernel, Proceedings of the 2nd Usenix Mach Symposium, Nov. 20–22, 199 pp. 233–250.

(List continued on next page.)

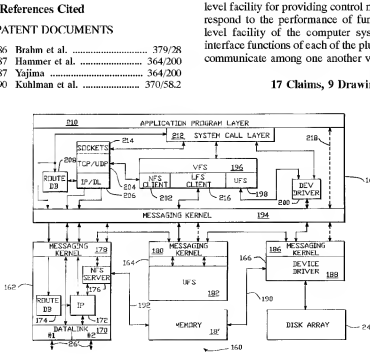
Primary Examiner—Lucien U. Toplu

Attorney, Agent, or Firm—Fliesler Dubb Meyer & Lovejoy

[57] **ABSTRACT**

This is achieved in a computer system employing a multiple facility operating system architecture. The computer system includes a plurality of processor units for implementing a predetermined set of peer-level facilities wherein each peer-level facility includes a plurality of related functions and a communications bus for interconnecting the processor units. Each of the processor units includes a central processor and the stored program that, upon execution, provides for the implementation of a predetermined peer-level facility of the predetermined set of peer-level facilities, and for performing a multi-tasking interface function. The multi-tasking interface function is responsive to control messages for selecting for execution functions of the predetermined peer-level facility and that is responsive to the predetermined peer-level facility for providing control messages to request or to respond to the performance of functions of another peer-level facility of the computer system. The multi-tasking interface functions of each of the plurality of processor units communicate among one another via the network bus.

17 Claims, 9 Drawing Sheets



U.S. PATENT DOCUMENTS

5,133,053	7/1992	Johnson et al.	395/200
5,218,697	6/1993	Chung	395/650
5,355,453	10/1994	Row et al.	395/200
5,506,988	4/1996	Weber et al.	395/650
5,557,798	9/1996	Skeen et al.	395/650

OTHER PUBLICATIONS

Barrera III, A Fast Mach Network IPC Implementation, Proceedings of the Usenix Mach Symposium, Nov. 1991.

Kupfer, Sprite on Mach, Proceeding of the 3rd Usenix Mach Symposium, Apr. 1993.

Roy, Unix File Access and Caching in a Multicomputer Environment, Proceedings of the Usenix Mach Symposium, Apr. 1993.

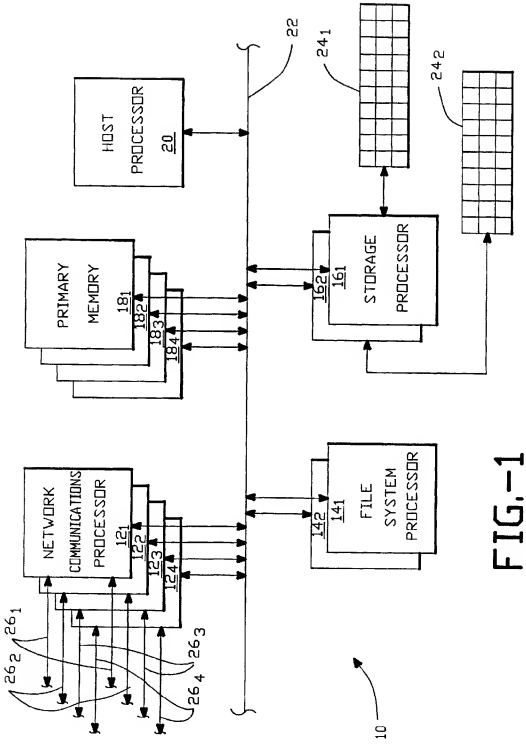

HOST PROCESSOR 20
22
241
242
PRIMARY MEMORY 181 182 183 184
STORAGE PROCESSOR 161 162
NETWORK COMMUNICATIONS PROCESSOR 121 122 123 124
FILE SYSTEM PROCESSOR 141 142
261 262 263 264
10

FIG.-1

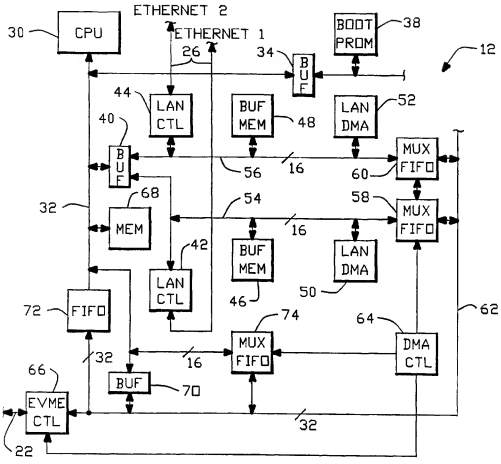

ETHERNET 2
ETHERNET 1
30
CPU
26
34
BUF
BOOT PROM
38
12
44
LAN CTL
BUF MEM
48
LAN DMA
52
40
BUF
56
16
60
MUX FIFO
32
68
MEM
54
58
MUX FIFO
16
42
BUF MEM
LAN DMA
LAN CTL
46
50
72
FIFO
74
64
62
32
16
MUX FIFO
DMA CTL
66
BUF
70
EVME CTL
32
22

FIG.-2

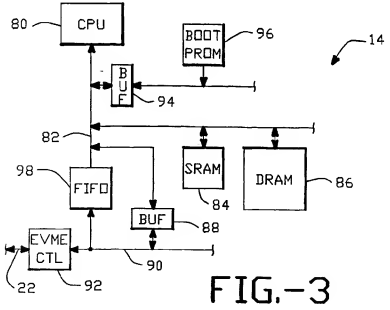

80
CPU
BOOT PROM
96
14
BUF
94
82
98
FIFO
SRAM
DRAM
86
84
BUF
88
EVME CTL
90
22
92

FIG.-3

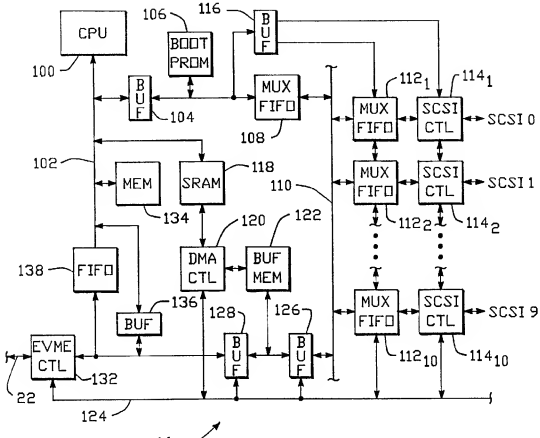

CPU
106
116
BUF
BOOT PROM
100
BUF
104
MUX FIFO
108
112 1
114 1
MUX FIFO
SCSI CTL
SCSI 0
102
MEM
SRAM
118
110
MUX FIFO
SCSI CTL
SCSI 1
134
120
122
112 2
114 2
138
FIFO
DMA CTL
BUF MEM
136
BUF
128
126
MUX FIFO
SCSI CTL
SCSI 9
EVME CTL
BUF
BUF
112 10
114 10
132
22
124
16

FIG.—4

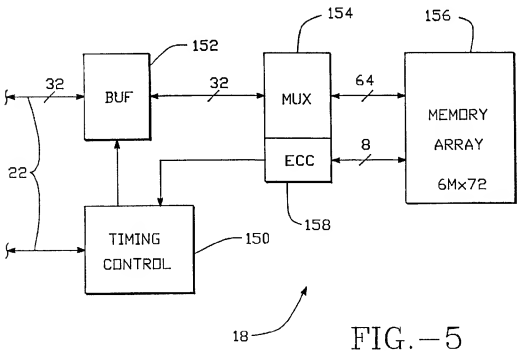

154
156
152
32
BUF
32
MUX
64
MEMORY ARRAY 6Mx72
22
ECC
8
158
TIMING CONTROL
150
18

FIG.—5

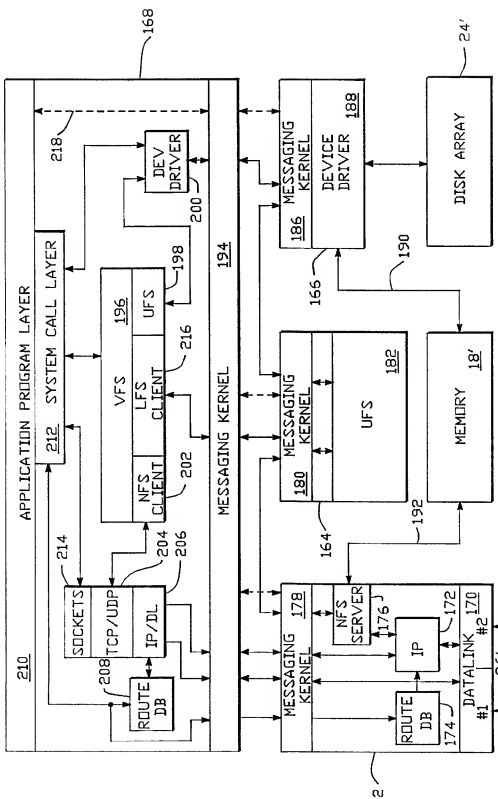

168
APPLICATION PROGRAM LAYER
210
SYSTEM CALL LAYER
212
218
DEV. DRIVER
200
198
VFS
196
UFS
LFS CLIENT
NFS CLIENT
202
216
MESSAGING KERNEL
194
SOCKETS
214
TCP/UDP
IP/DL
204
206
208
ROUTE DB
MESSAGING KERNEL
178
NFS SERVER
176
IP
172
DATALINK #1
DATALINK #2
170
26
ROUTE DB
174
162
MESSAGING KERNEL
180
UFS
182
164
192
MEMORY
18'
160
MESSAGING KERNEL
186
DEVICE DRIVER
188
166
190
DISK ARRAY
24'

FIG.—6

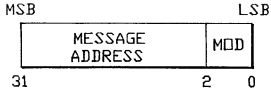


FIG.—7

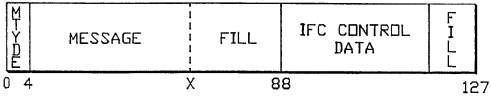


FIG.—8

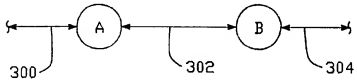


FIG.—9

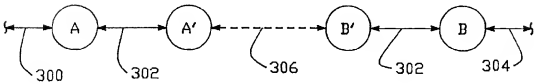


FIG.—10

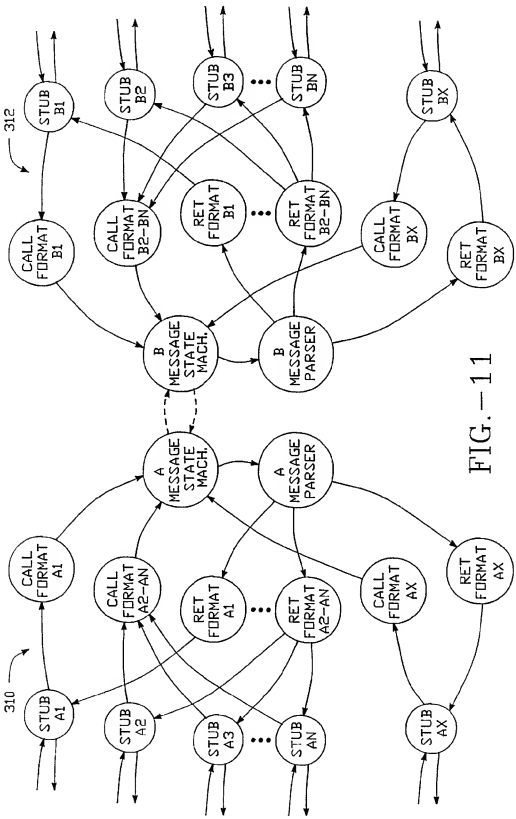

312
310
STUB B1
STUB B2
STUB B3
STUB BN
STUB BX
CALL FORMAT B1
CALL FORMAT B2-BN
RET FORMAT B1
RET FORMAT B2-BN
CALL FORMAT BX
RET FORMAT BX
B MESSAGE STATE MACH.
B MESSAGE PARSER
A MESSAGE STATE MACH.
A MESSAGE PARSER
CALL FORMAT A1
CALL FORMAT A2-AN
RET FORMAT A1
RET FORMAT A2-AN
CALL FORMAT AX
RET FORMAT AX
STUB A1
STUB A2
STUB A3
STUB AN
STUB AX

FIG. — 11

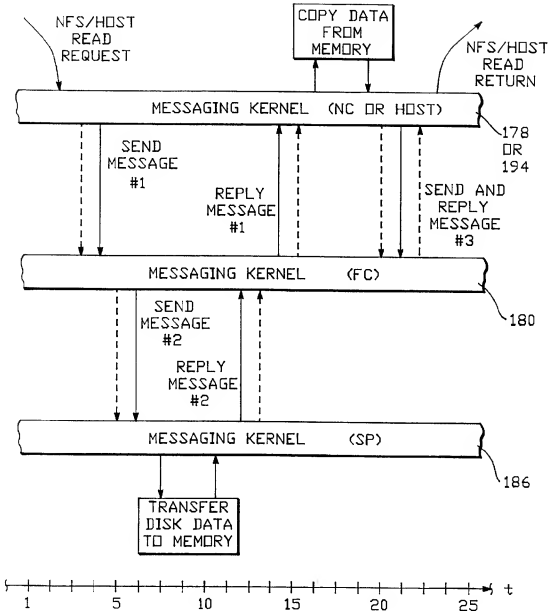

NFS/HOST READ REQUEST
COPY DATA FROM MEMORY
NFS/HOST READ RETURN
MESSAGING KERNEL (NC OR HOST)
178 OR 194
SEND MESSAGE #1
REPLY MESSAGE #1
SEND AND REPLY MESSAGE #3
MESSAGING KERNEL (FC)
180
SEND MESSAGE #2
REPLY MESSAGE #2
MESSAGING KERNEL (SP)
186
TRANSFER DISK DATA TO MEMORY
1
5
10
15
20
25
t

FIG.—12

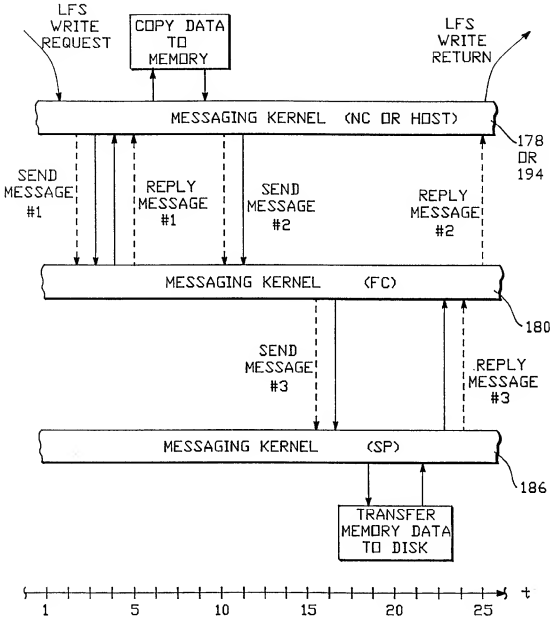

LFS WRITE REQUEST
COPY DATA TO MEMORY
LFS WRITE RETURN
MESSAGING KERNEL (NC OR HOST)
178 OR 194
SEND MESSAGE #1
REPLY MESSAGE #1
SEND MESSAGE #2
REPLY MESSAGE #2
MESSAGING KERNEL (FC)
180
SEND MESSAGE #3
REPLY MESSAGE #3
MESSAGING KERNEL (SP)
186
TRANSFER MEMORY DATA TO DISK
1
5
10
15
20
25
t

FIG.—13

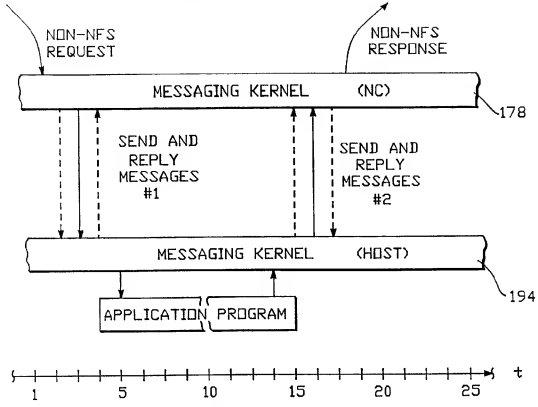

NON-NFS REQUEST
NON-NFS RESPONSE
MESSAGING KERNEL (NC)
178
SEND AND REPLY MESSAGES #1
SEND AND REPLY MESSAGES #2
MESSAGING KERNEL (HOST)
194
APPLICATION PROGRAM
t
1
5
10
15
20
25

FIG.—14

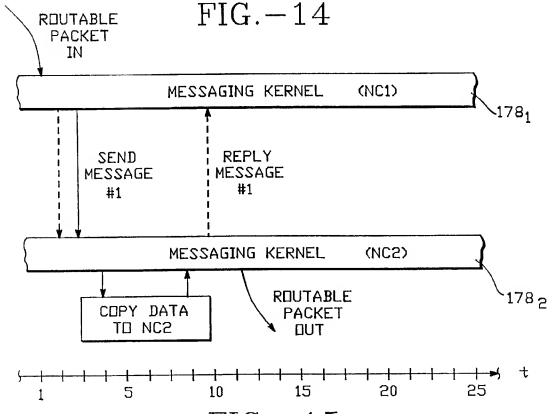

ROUTABLE PACKET IN
MESSAGING KERNEL (NC1)
178 1
SEND MESSAGE #1
REPLY MESSAGE #1
MESSAGING KERNEL (NC2)
178 2
COPY DATA TO NC2
ROUTABLE PACKET OUT
t
1
5
10
15
20
25

FIG.—15

MULTIPLE SOFTWARE-FACILITY COMPONENT OPERATING SYSTEM FOR CO-OPERATIVE PROCESSOR CONTROL WITHIN A MULTIPROCESSOR COMPUTER SYSTEM

This is a division of U.S. patent application Ser. No. 08/225,356, filed Apr. 8, 1994, now U.S. Pat. No. 5,485,579, which is a continuation of U.S. patent application Ser. No. 07/875,585, filed Apr. 28, 1992, abandoned, which is a continuation of Ser. No. 07/404,885, filed Sep. 8, 1989, abandoned.

CROSS-REFERENCE TO RELATED APPLICATIONS

The present application is related to the following U.S. Patent Applications:

1. PARALLEL I/O NETWORK FILE SERVER ARCHITECTURE, inventors: John Row, Larry Boucher, William Pitts, and Steve Blightman;
2. ENHANCED VMEBUS PROTOCOL UTILIZING SYNCHRONOUS HANDSHAKING AND BLOCK MODE DATA TRANSFER, inventor: Daryl D. Starr;
2. BUS LOCKING FIFO MULTI-PROCESSOR COMMUNICATIONS SYSTEM UTILIZING PSEUDO-SYNCHRONOUS HANDSHAKING AND BLOCK MODE DATA TRANSFER invented by William Pitts, Stephen Blightman and Daryl D. Starr;
3. IMPROVED FAST TRANSFER DIRECT MEMORY ACCESS CONTROLLER, invented by Daryl Starr, Stephen Blightman and Larry Boucher.

The above applications are all assigned to the assignee of the present invention and are all expressly incorporated herein by reference.

1. Field of the Invention

The present invention is generally related to operating system software architectures and, in particular, to a multi-processor operating system architecture based on multiple independent multi-tasking process kernels.

2. Background of the Invention

The desire to improve productivity, in circumstances involving computers, is often realized by an improvement in computing throughput. Conventional file servers are recognized as being a limiting factor in the potential productivity associated with their client workstations.

A file server is typically a conventional computer system coupled through a communications network, such as Ethernet, to client workstations and potentially other workstation file servers. The file server operates to provide a common resource base to its clients. The primary resource is typically the central storage and management of data files, but additional services including single point execution of certain types of programs, electronic mail delivery and gateway connection to other file servers and services are generally also provided.

The client workstations may utilize any of a number of communication network protocols to interact with the file server. Perhaps the most commonly known, if not most widely used, protocol suite is TCP/IP. This protocol suite and its supporting utility programs, provide for the creation of logical communication channels between multiple client workstations and a file server. These communication channels are generally optimized for point-to-point file transfers, i.e., multi-user file access control or activity administration is not provided. In addition, the supporting utility programs for these protocols impose a significant degree of user

interaction in order to initiate file transfers as well as the entire responsibility to manage the files once transferred.

Recently, a number of network connected remote file system mechanisms has been developed to provide clients with a single consistent view of a file system of data files, even though portions of the file system may be physically distributed between a client's own local storage, one or more file servers or even other client workstations. These network file system mechanisms operate to hide the distinction between local data files and data files in the remotely distributed portions of the file system accessible only through the network. The advantages of such file system mechanisms include retention of multi-user access controls over the data files physically present on the server, to the extent intrinsically provided by a server, and a substantial simplification of a client workstation's view and productive utilization of the file system.

Two implementations of a network file system mechanism are known as the network file system (NFS), available from Sun Microsystems, Inc., and the remote file sharing (RFS) system available from American Telephone and Telegraph, Inc.

The immediate consequence of network file system mechanism is that they have served to substantially increase the throughput requirements of the file server itself, as well as that of the communications network. Thus, the number of client workstations that can be served by a single file server must be balanced against the reduction in productivity resulting from increased file access response time and the potentially broader effects of a degradation in communication efficiency due to the network operating at or above its service maximum.

An increase in the number of client workstations is conventionally handled by the addition of another file server, duplicating or possibly partitioning the file system between the file servers, and providing a dedicated high bandwidth network connection between the file servers. Thus, another consequence of the limited throughput of conventional file servers is a greater cost and configuration complexity of the file server base in relation to the number of client workstations that can be effectively serviced.

Another complicating factor, for many technical and practical reasons, is a requirement that the file server be capable of executing the same or a similar operating system as the attached client workstations. The reasons include the need to execute maintenance and monitoring programs on the file server, and to execute programs, such as database servers, that would excessively load the communications network if executed remotely from the required file data. Another often overlooked consideration is the need to avoid the cost of supporting an operating system that is unique to the file server.

Given these considerations, the file server is typically only a conventional general purpose computer with an extended data storage capacity and communications network interface that is little different from that present on each of the client workstations. Indeed, many file servers are no more than a physically repackaged workstation. Unfortunately, even with multiple communications network interfaces, such workstation-based computers are either incapable or inappropriate, from a cost/performance viewpoint, to perform as a single file server to a large group of client workstations.

The throughput offered by conventional general purpose computers, considered in terms of their sustained file system facility data transfer bandwidth potential, is limited by a number of factors, though primarily due to the general

purpose nature of their design. Computer system design is necessarily dependent on the level and nature of the operating system to be executed, the nature of the application load to be executed, and the degree of homogeneity of applications. For example, a computer system utilized solely for scientific computations may forego an operating system entirely, may be restricted to a single user at a time, and employ specialized computation hardware optimized for the anticipated highly homogeneous applications. Conversely, where an operating system is required, the system design typically calls for the utilization of dedicated peripheral controllers, operated under the control of a single processor executing the operating system, in an effort to reduce the peripheral control processing overhead of the system's single primary processor. Such is the design of most conventional file servers.

A recurring theme in the design of general purpose computer systems is to increase the number of active primary processors. In the simplest analysis, a linear improvement in the throughput performance of the computer system might be expected. However, utilization of increasing numbers of primary processors is typically thwarted by the greater growth of control overhead and contention for common peripheral resources. Indeed, the net improvement in throughput is often seen to increase slightly before declining rapidly as the number of processors is increased.

SUMMARY OF THE INVENTION

Therefore, a general purpose of the present invention is to provide an operating system architecture for the control of a multi-processor system to provide an efficient, expandable computer system for servicing network file system requests.

This is achieved in a computer system employing a multiple facility operating system architecture. The computer system includes a plurality of processor units for implementing a predetermined set of peer-level facilities, wherein each peer-level facility implements a plurality of related functions, and a communications bus for interconnecting the processor units. Each of the processor units includes a central processor and a stored program that, upon execution, provides for the implementation of a predetermined peer-level facility and for implementing a multi-tasking interface function. The multi-tasking interface function is responsive to control messages for selecting for execution functions of the predetermined peer-level facility. The multi-tasking interface function is also responsive to the predetermined peer-level facility for providing control messages to request or to respond to the performance of functions of another peer-level facility of the computer system. The multi-tasking interface functions of each of the plurality of processor units communicate among one another via the network bus.

Thus, in a preferred embodiment of the present invention, the set of peer-level facilities includes network communications, file system control, storage control and a local host operating system.

An advantage of the present invention is that it provides for the implementation of multiple facilities, each instance on a respective processor, all within a single cohesive system while incurring little additional control overhead in order to maintain operational coherency.

Another advantage of the present invention is that direct peer to peer-level facility communication is supported in order to minimize overhead in processing network file system requests.

A further advantage of the present invention is that it realizes a computer system software architecture that is readily expandable to include multiple instances of each peer-level facility, and respective peer-level processors, in a single cohesive operating system environment including direct peer to peer-level facility communications between like facilities.

Yet another advantage of the present invention is that it may include an operating system as a facility operating concurrently and without conflict with the otherwise independent peer to peer-level facility communications of the other peer-level facilities. The operating system peer-level facility may itself be a conventional operating system suitably compatible with the workstation operating systems so as to maintain compatibility with "standard" file server operating systems. The operating system peer-level facility may be used to handle exception conditions from the other peer-level facilities including handling of non-network file system requests. Consequently, the multiple facility operating system architecture of the present invention appears to client workstations as a conventional, single processor file server.

A still further advantage of the present invention is that it provides a message-based operating system architecture framework for the support of multiple, specialized peer-level facilities within a single cohesive computer system; a capability particularly adaptable for implementation of a high-performance, high-throughput file server.

BRIEF DESCRIPTION OF THE DRAWINGS

These and other attendant advantages and features of the present invention will become apparent and readily appreciated as the same becomes better understood by reference to the following detailed description when considered in conjunction with the accompanying drawings, in which like reference numerals indicate like parts throughout the figures thereof, and wherein:

FIG. **1** is a simplified block diagram of a preferred computer system architecture for implementing the multiple facility operating system architecture of the present invention;

FIG. **2** is a block diagram of a network communications processor suitable for implementing a network communications peer-level facility in accordance with a preferred embodiment of the present invention;

FIG. **3** is a block diagram of a file system processor suitable for implementing a file system controller peer-level facility in accordance with a preferred embodiment of the present invention;

FIG. **4** is a block diagram of a storage processor suitable for implementing a storage peer-level facility in accordance with a preferred embodiment of the present invention;

FIG. **5** is simplified block diagram of a primary memory array suitable for use as a shared memory store in a preferred embodiment of the present invention;

FIG. **6** is a block diagram of the multiple facility operating system architecture configured in accordance with a preferred embodiment of the present invention;

FIG. **7** is a representation of a message descriptor passed between peer-level facilities to identify the location of a message;

FIG. **8** is a representation of a peer-level facility message as used in a preferred embodiment of the present invention;

FIG. **9** is a simplified representation of a conventional program function call;

FIG. **10** is a simplified representation of an inter-facility function call in accordance with the preferred embodiment of the present invention;

FIG. **11** is a control state diagram illustrating the interface functions of two peer-level facilities in accordance with a preferred embodiment of the present invention;

FIG. **12** is an illustration of a data flow for an LFS read request through the peer-level facilities of a preferred embodiment of the present invention;

FIG. **13** is an illustration of a data flow for an LFS write request through the peer-level facilities of a preferred embodiment of the present invention;

FIG. **14** illustrates the data flow of a non-LFS data packet between the network communication and local host peer-level facilities in accordance with a preferred embodiment of the present invention; and

FIG. **15** illustrates the data flow of a data packet routed between two network communications peer-level facilities in accordance with a preferred embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

While the present invention is broadly applicable to a wide variety of hardware architectures, and its software architecture may be represented and implemented in a variety of specific manners, the present invention may be best understood from an understanding of its preferred embodiment.

I. System Architecture Overview

A. Hardware Architecture Overview

A block diagram representing the preferred embodiment of the hardware support for the present invention, generally indicated by the reference numeral **10**, is provided in FIG. **1**. The architecture of the preferred hardware system **10** is described in the above-identified related application entitled PARALLEL I/O NETWORK FILE SERVER ARCHITECTURE; which application is expressly incorporated herein by reference.

The hardware components of the system **10** include multiple instances of network controllers **12**, file system controllers **14**, and mass storage processors, **16**, interconnected by a high-bandwidth backplane bus **22**. Each of these controllers **12**, **14**, **16** preferably include a high performance processor and local program store, thereby minimizing their need to access the bus **22**. Rather, bus **22** accesses by the controllers **12**, **14**, **16** are substantially limited to transfer accesses as required to transfer control information and client workstation data between the controllers **12**, **14**, **16** system memory **18**, and a local host processor **20**, when necessary.

The illustrated preferred system **10** configuration includes four network controllers $\mathbf{12}_{1\text{-}4}$, two file controllers $\mathbf{14}_{1\text{-}2}$, two mass storage processors $\mathbf{16}_{1\text{-}2}$, a bank of four system memory cards $\mathbf{18}_{1\text{-}4}$, and a host processor **20** coupled to the backplane bus **22**. The invention, however, is not limited to this number and type of processors. Rather, six or more network communications processors **12** and two or more host processors **20** could be implemented within the scope of the present invention.

Each network communications processor (NP) $\mathbf{12}_{1\text{-}4}$ preferably includes a Motorola 68020 processor for supporting two independent Ethernet network connections, shown as the network pairs $\mathbf{26}_1$–$\mathbf{26}_4$. Each of the network connections directly support the ten megabit per second data rate specified for a conventional individual Ethernet network connection. The preferred hardware embodiment of the present invention thus realizes a combined maximum data throughput potential of 80 megabits per second.

The file system processors (FP) $\mathbf{14}_{1\text{-}2}$, intended to operate primarily as a specialized compute engines, each include a high-performance Motorola 68020 based microprocessor, four megabytes of local data store and a smaller quarter-megabyte high-speed program memory store.

The storage processors (SP) $\mathbf{16}_{1\text{-}2}$ function as intelligent small computer system interface (SCSI) controllers. Each includes a Motorola 68020 micro-processor, a local program and data memory, and an array of ten parallel SCSI channels. Drive arrays $\mathbf{24}_{1\text{-}2}$ are coupled to the storage processors $\mathbf{16}_{1\text{-}2}$ to provide mass storage. Preferably, the drive arrays $\mathbf{24}_{1\text{-}2}$ are ten unit-wide arrays of SCSI storage devices uniformly from one to three units deep. The preferred embodiment of the present invention uses conventional 768 megabyte 5¼-inch hard disk drives for each unit of the arrays $\mathbf{24}_{1\text{-}2}$. Thus, each drive array level achieves a storage capacity of approximately 6 gigabytes, with each storage processor readily supporting a total of 18 gigabytes. Consequently, a system **10** is capable of realizing a total combined data storage capacity of 36 gigabytes.

The local host processor **20**, in the preferred embodiments of the present invention, is a Sun central processor card, model Sun 3E120, manufactured and distributed by Sun Microsystems, Inc.

Finally, the system memory cards **18** each provide 48 megabytes of 32-bit memory for shared use within the computer system **10**. The memory is logically visible to each of the processors of the system **10**.

A VME bus **22** is used in the preferred embodiments of the present invention to interconnect the network communication processors **12**, file system processors **14**, storage processors **16**, primary memory **18**, and host processor **20**. The hardware control logic for controlling the VME bus **22**, at least as implemented on the network communication processor **12** and storage processor **16**, implements a bus master fast transfer protocol in addition to the conventional VME transfer protocols. The system memory **18** correspondingly implements a modified slave VME bus control logic to allow the system memory **18** to also act as the fast data transfer data source or destination for the network communication processors **12** and storage processors **16**. The fast transfer protocol is described in the above-identified related application entitled "ENHANCED VMEBUS PROTOCOL UTILIZING SYNCHRONOUS HANDSHAKING AND BLOCK MODE DATA TRANSFER"; which application is expressly incorporated herein by reference.

It should be understood that, while the system **10** configuration represents the initially preferred maximum hardware configuration, the present invention is not limited to the preferred number or type of controllers, the preferred size and type of disk drives or use of the preferred fast data transfer VME protocol.

B. Software Architecture Overview

Although applicable to a wide variety of primary, or full function, operating systems such as MVS and VMS, the preferred embodiment of the present invention is premised on the Unix operating system as distributed under license by American Telephone and Telegraph, Inc. and specifically the SunOS version of the Unix operating system, as available from Sun Microsystems, Inc. The architecture of the Unix operating system has been the subject of substantial academic study and many published works including "The Design of the Unix Operating System", Maurice J. Bach, Prentice Hall, Inc., 1986.

In brief, the Unix operating system is organized around a non-preemptive, multi-tasking, multi-user kernel that implements a simple file-oriented conceptual model of a file

system. Central to the model is a virtual file system (VFS) interface that operates to provide a uniform file oriented, multiple file system environment for both local and remote files.

Connected to the virtual file system is the Unix file system (UFS). The UFS allows physical devices, pseudo-devices and other logical devices to appear and be treated, from a client's perspective, as simple files within the file system model. The UFS interfaces to the VFS to receive and respond to file oriented requests such as to obtain the attributes of a file, the stored parameters of a physical or logical device, and, of course, to read and write data. In carrying out these functions, the UFS interacts with a low level software device driver that is directly responsible for an attached physical mass storage device. The UFS handles all operations necessary to resolve logical file oriented operations, as passed from the VFS, down to the level of a logical disk sector read or write request.

The VFS, in order to integrate access to remote files into the file system model, provides a connection point for network communications through the network file system mechanism, if available. The preferred network file system mechanism, NFS, is itself premised on the existence of a series of communication protocol layers that, inclusive of NFS and within the context of the present invention, can be referred to as an NFS stack. These layers, in addition to an NFS "layer," typically include a series of protocol handling layers generally consistent with the International Standards Organization's Open Systems Interconnection (ISO/OSI) model. The OSI model has been the subject of many publications, both regarding the conceptual aspects of the model as well as specific implementations, including "Computer Networks, 2nd Edition", Andrew S. Tanenbaum, Prentice Hall, 1988.

In summary, the OSI layers utilized by the present invention include all seven layers described in the OSI reference model: application, presentation, session, transport, network, data link and physical layers. These layers are summarized below, in terms of their general purpose, function and implementation for purposes of the present invention.

The application layer protocol, NFS, provides a set of remote procedure call definitions, for use in both server and client oriented contexts, to provide network file services. As such, the NFS layer provides a link between the VFS of the Unix kernel and the presentation protocol layer.

The presentation layer protocol, provided as an external data representation (XDR) layer, defines a common description and encoding of data as necessary to allow transfer of data between different computer architectures. The XDR is thus responsible for syntax and semantic translation between the data representations of heterogeneous computer systems.

The session layer protocol, implemented as a remote procedure call (RPC) layer, provides a remote procedure call capability between a client process and a server process. In a conventional file server, the NFS layer connects through the XDR layer to the RPC layer in a server context to support the file oriented data transfers and related requests of a network client.

The transport layer protocol, typically implemented as either a user datagram protocol (UDP) or transmission control protocol (TCP) layer, provides for a simple connectionless datagram delivery service. NFS uses UDP.

The network layer protocol, implemented as an internet protocol (IP) layer, performs internet routing, based on address mappings stored in an IP routing database, and data packet fragmentation and reassembly.

The data link (DL) layer manages the transfer and receipt of data packets based on packet frame information. Often this layer is referred to as a device driver, since it contains the low level software control interface to the specific communications hardware, including program control of low level data transmission error correction/handling and data flow control. As such, it presents a hardware independent interface to the IP layer.

Finally, the physical layer, an Ethernet controller, provides a hardware interface to the network physical transmission medium.

The conventional NFS stack, as implemented for the uniprocessor VAX architecture, is available in source code form under license from Sun Microsystems, Inc.

The preferred embodiment of the present invention utilizes the conventional SunOS Unix kernel, the Sun/VAX reference release of the UFS, and the Sun/VAX reference release of the NFS stack as its operating system platform. The present invention establishes an instantiation of the NFS stack as an independent, i.e., separately executed, software entity separate from the Unix kernel. Instantiations of the UFS and the mass storage device driver are also established as respective independent software entities, again separate from the Unix kernel. These entities, or peer-level facilities, are each provided with an interface that supports direct communication between one another. This interface, or messaging kernel layer, includes a message passing, multi-tasking kernel. The messaging kernel's layers are tailored to each type of peer-level facility in order to support the specific facility's functions. The provision for multi-tasking operation allows the peer-level facilities to manage multiple concurrent processes. Messages are directed to other peer-level facilities based upon the nature of the function requested. Thus, for NFS file system requests, request messages may be passed from an NFS network communications peer-level facility directly to a UFS file system peer-level facility and, as necessary, then to the mass storage peer-level facility. The relevant data path is between the NFS network communications peer-level facility and the mass storage peer-level facility by way of the VME shared address space primary memory. Consequently, the number of peer-level facilities is not logically bounded and servicing of the most common type of client workstation file system needs is satisfied while requiring only a minimum amount of processing.

Finally, a Unix kernel, including its own NFS stack, UFS, and mass storage device driver, is established as a another peer-level facility. As with the other peer-level facilities, this operating system facility is provided with a multi-tasking interface for interacting concurrently with the other peer-level facilities as just another entity within the system **10**. While the operating system kernel peer-level facility is not involved in the immediate servicing of most NFS requests, it interacts with the NFS stack peer-level facility to perform general management of the ARP and IP data bases, the initial NFS file system access requests from a client workstation, and to handle any non-NFS type requests that might be received by the NFS stack peer-level facility.

II. Peer-level Processors

A. Network Control Processor

A block diagram of the preferred network control processor is shown in FIG. **2**. The network controller **12** includes a 32-bit central processing unit (CPU) **30** coupled to a local CPU bus **32** that includes address, control and data lines. The CPU is preferably a Motorola 68020 processor. The data line portion of the CPU bus **32** is 32 bits wide. All of the elements coupled to the local bus **32** of the network con-

troller **12** are memory mapped from the perspective of the CPU **30**. This is enabled by a buffer **34** that connects the local bus **32** to a boot PROM **38**. The boot PROM **38** is utilized to store a boot program and its necessary start-up and operating parameters. Another buffer **40** allows the CPU **30** to separately address a pair of Ethernet local area network (LAN) controllers **42**, **44**, their local data packet memories **46**, **48**, and their associated packet direct memory access (DMA) controllers **50**, **52**, via two parallel address, control, and 16-bit wide data buses **54**, **56**. The LAN controllers **42**, **44** are programmed by the CPU **30** to utilize their respective local buffer memories **46**, **48** for the storage and retrieval of data packets as transferred via the Ethernet connections **26**. The DMA controllers **50**, **52** are programmed by the CPU **30** to transfer data packets between the buffer memories **46**, **48** and a respective pair of multiplexing FIFOs **58**, **60** also connected to the LAN buses **54**, **56**. The multiplexing FIFOs **58**, **60** each include a 16-bit to 32-bit wide data multiplexer/demultiplexer, coupled to the data portion of the LAN buses **54**, **56**, and a pair of internal FIFO buffers. Thus, for example in the preferred embodiment of the present invention, a first 32-bit wide internal FIFO is coupled through the multiplexer to the 16-bit wide LAN bus **54**. The second internal FIFO, also 32-bit wide, is coupled to a secondary data bus **62**. These internal FIFO buffers of the multiplexing FIFO **58**, as well as those of the multiplexing FIFO **60**, may be swapped between their logical connections to the LAN buses, **54**, **56** and the secondary data bus **62**. Thus, a large difference in the data transfer rate of the LAN buses **54**, **60** and the secondary data bus **62** can be maintained for a burst data length equal to the depth of the internal FIFOs **58**, **60**.

A high speed DMA controller **64**, controlled by the CPU **30**, is provided to direct the operation of the multiplexing FIFOs **58**, **60** as well as an enhanced VME control logic block **66**, through which the data provided on the secondary data bus **62** is communicated to the data lines of the VME bus **22**. The purpose of the multiplexing FIFOs **58**, **60**, besides acting as a 16-bit to 32-bit multiplexer and buffer, is to ultimately support the data transfer rate of the fast transfer mode of the enhanced VME control logic block **66**.

Also connected to the local CPU data bus **32** is a quarter megabyte block of local shared memory **68**, a buffer **70**, and a third multiplexing FIFO **74**. The memory **68** is shared in the sense that it also appears within the memory address space of the enhanced VME bus **22** by way of the enhanced VME control logic block **66** and buffer **70**. The buffer **70** preferably provides a bidirectional data path for transferring data between the secondary data bus **62** and the local CPU bus **32** and also includes a status register array for receiving and storing status words either from the CPU **30** or from the enhanced VME bus **22**. The multiplexing FIFO **74**, identical to the multiplexing FIFOs **58**, **60**, provides a higher speed, block-oriented data transfer capability for the CPU **30**.

Finally, a message descriptor FIFO **72** is connected between the secondary data bus **62** and the local CPU bus **32**. Preferably, the message descriptor FIFO **72** is addressed from the enhanced VME bus **22** as a single shared memory location for the receipt of message descriptors. Preferably the message descriptor FIFO **72** is 32-bit wide, single buffer FIFO with a 256-word storage capability. In accordance with the preferred embodiments of the present invention, the message descriptor FIFO is described in detail in the above-referenced related application "BUS LOCKING FIFO MULTI-PROCESSOR COMMUNICATIONS SYSTEM"; which application is hereby incorporated by reference. However, for purposes of completeness, an enhancement embodied in the enhanced VME control logic block **66** is that it preemptively allows writes to the message descriptor FIFO **72** from the enhanced VME bus **22** unless the FIFO **72** is full. Where a write to the message descriptor FIFO **72** cannot be accepted, the enhanced VME control logic block **66** immediately declines the write by issuing a VME bus error signal onto the enhanced VME bus.

B. File System Control Processor

The preferred architecture of a file system processor **14** **60** is shown in FIG. **3**. A CPU **80**, preferably a Motorola 68020 processor, is connected via a local CPU address, control and 32-bit wide data bus **82** to the various elements of the file controller **14**. These principle elements include a 256 kilo-bytes of static RAM block **84**, used for storing the file system control program, and a four megabyte dynamic RAM block **86** for storing local data, both connected directly to the local CPU bus **82**. A buffer **88** couples the local CPU bus **82** to a secondary 32-bit wide data bus **90** that is, in turn, coupled through an enhanced VME control and logic block **92** to the data bus lines of the VME bus **22**. In addition to providing status register array storage, the buffer **88** allows the memory blocks **84**, **86** to be accessible as local shared memory on the VME bus **22**. A second buffer **94** is provided to logically position a boot PROM **96**, containing the file controller initialization program, within the memory address map of the CPU **80**. Finally, a single buffer message descriptor FIFO **98** is provided between the secondary data bus **90** and the local CPU bus **82**. The message descriptor FIFO **98** is again provided to allow preemptive writes to the file controller **14** from the enhanced VME bus **22**.

C. Storage Control Processor

A block diagram of a storage processor **16** is provided in FIG. **4**. A CPU **100**, preferably a Motorola 68020 processor, is coupled through a local CPU address, control and 32-bit wide data bus **102** and a buffer **104** to obtain access to a boot PROM **106** and a double-buffered multiplexing FIFO **108** that is, in turn, connected to an internal peripheral data bus **110**. The internal peripheral data bus **110** is, in turn, coupled through a parallel channel array of double-buffered multiplexing FIFOs $112_{1\text{-}10}$ and SCSI channel controllers $114_{1\text{-}10}$. The SCSI controllers $114_{1\text{-}10}$ support the respective SCSI buses (SCSI**0**-SCSI**9**) that connect to a drive array **24**.

Control over the operation of the double buffer FIFO $112_{1\text{-}10}$ and SCSI controller $114_{1\text{-}10}$ arrays is ultimately by the CPU **100** via a memory-mapped buffer **116** and a first port of a dual ported SRAM command block **118**. The second port of the SRAM block **118** is coupled to a DMA controller **120** that controls the low level transfer of data between the double-buffered FIFOs **108**, $112_{1\text{-}10}$, a temporary store buffer memory **122** and the enhanced VME bus **22**. In accordance with a preferred embodiment of the present invention, the DMA controller responds to commands posted by the CPU **100** in the dual-ported SRAM block **118** to select any of the double-buffered FIFOs **108**, $112_{1\text{-}10}$, the buffer memory **122**, and the enhanced VME bus **22** as a source or destination of a data block transfer. To accomplish this, the DMA controller **120** is coupled through a control bus **124** to the double buffered FIFOs **108**, $112_{1\text{-}10}$, the SCSI controllers $114_{1\text{-}10}$, the buffer memory **122**, a pair of secondary data bus buffers **126**, **128**, and an enhanced VME control and logic block **132**. The buffers **126**, **128** are used to route data by selectively coupling the internal peripheral data bus **110** to a secondary data bus **130** and the buffer memory **122**. The DMA controller **120**, as implemented in accordance with a preferred embodiment of the present invention, is described in detail in the above-referenced related application "IMPROVED FAST TRANSFER DIRECT MEMORY ACCESS CONTROLLER";

which application is hereby incorporated by reference. Finally, a one megabyte local shared memory block **134**, a high speed buffer and register array **136**, and a preemptive write message descriptor FIFO **138** are provided connected directly to the local CPU data bus **102**. The buffer **136** is also coupled to the secondary data bus **130**, while the message descriptor FIFO **138** is coupled to the secondary data bus **130**.

D. Primary Memory Array

FIG. **5** provides a simplified block diagram of the preferred architecture of a memory card **18**. Each memory card **18** operates as a slave on the enhanced VME bus and therefore requires no on-board CPU. Rather, a timing control block **150** is sufficient to provide the necessary slave control operations. In particular, the timing control block **150**, in response to control signals from the control portion of the enhanced VME bus **22** enables a 32-bit wide buffer **152** for an appropriate direction transfer of 32-bit data between the enhanced VME bus **22** and a multiplexer unit **154**. The multiplexer **154** provides a multiplexing and demultiplexing function, depending on data transfer direction, for a six megabit by seventy-two bit word memory array **156**. An error correction code (ECC) generation and testing unit **158** is coupled to the multiplexer **154** to generate or verify, again depending on transfer direction, eight bits of ECC data per memory array word. The status of each ECC verification operation is provided back to the timing control block **150**.

E. Host Processor

The host processor **20**, as shown in FIG. **1**, is a conventional Sun 3E120 processor. Due to the conventional design of this product, a software emulation of a message descriptor FIFO is performed in a reserved portion of the local host processor's shared memory space. This software message descriptor FIFO is intended to provide the functionality of the message descriptor FIFOs **72**, **98**, and **138**. A preferred embodiment of the present invention includes a local host processor **20'**, not shown, that includes a hardware preemptive write message descriptor FIFO, but that is otherwise functionally equivalent to the processor **20**.

III. Peer-level Facility Architecture

A. Peer-Level Facility Functions

FIG. **6** provides an illustration of the multiple peer-level facility architecture of the present invention. However, only single instantiations of the preferred set of the peer-level facilities are shown for purposes of clarity.

The peer-level facilities include the network communications facility (NC) **162**, file system facility (FS) **164**, storage facility (S) **166** and host facility (H) **168**. For completeness, the memory **18** is illustrated as a logical resource **18'** and, similarly, the disk array **24** as a resource **24'**.

The network communications facility **162** includes a messaging kernel layer **178** and an NFS stack. The messaging kernel layer **178** includes a multi-tasking kernel that supports multiple processes. Logically concurrent executions of the code making up the NFS stack are supported by reference to the process context in which execution by the peer-level processor is performed. Each process is uniquely identified by a process ID (PID). Context execution switches by the peer-level processor are controlled by a process scheduler embedded in the facility's multi-tasking kernel. A process may be "active"—at a minimum, where process execution by the peer-level processor continues until a resource or condition required for continued execution is unavailable. A process is "blocked" when waiting for notice of availability of such resource or condition. For the network communications facility **162**, within the general context of the present invention, the primary source of process blocking is in the network and lower layers where a NC process will wait, executing briefly upon receipt of each of a series of packet frames, until sufficient packet frames are received to be assembled into a complete datagram transferrable to a higher level layer. At the opposite extreme, a NC process will block upon requesting a file system or local host function to be performed, i.e., any function controlled or implemented by another peer-level facility.

The messaging kernel layer **178**, like all of the messaging kernel layers of the present invention, allocates processes to handle respective communication transactions. In allocating a process, the messaging kernel layer **178** transfers a previously blocked process, from a queue of such processes, to a queue of active processes scheduled for execution by the multi-tasking kernel. At the conclusion of a communication transaction, a process is deallocated by returning the process to the queue of blocked processes.

As a new communication transaction is initiated, an address or process ID of an allocated process becomes the distinguishing datum by which the subsequent transactions are correlated to the relevant, i.e., proper handling, process. For example, where a client workstation initiates a new communication transaction, it provides its Ethernet address. The network communication facility, will store and subsequently, in responding to the request, utilize the client's Ethernet address to direct the response back to the specific requesting client.

The NC facility similarly provides a unique facility ID and the PID of its relevant process to another peer-level facility as part of any request necessary to complete a client's request. Thus, an NC facility process may block with certainty that the responding peer-level facility can direct its response back to the relevant process of the network communications peer-level facility.

The network and lower level layers of the NFS stack necessary to support the logical Ethernet connections **26'** are generally illustrated together as an IP layer **172** and data link layer **170**. The IP layer **172**, coupled to the IP route database **174**, is used to initially distinguish between NFS and non-NFS client requests. NFS requests are communicated to an NFS server **176** that includes the remaining layers of the NFS stack. The NFS server **176**, in turn, communicates NFS requests to the network communications messaging kernel layer **178**. By the nature of the call, the messaging kernel layer **178** is able to discern between NFS request calls, non-NFS calls from the IP layer **172** and network calls received directly from the network layers **170**.

For the specific instance of NFS requests, making up the large majority of requests handled by the network communications facility **162**, the relevant NC process calls the messaging kernel layer **178** to issue a corresponding message to the messaging kernel layer **180** of the file system facility **164**. The relevant NC process is blocked pending a reply message and, possibly, a data transfer. That is, when the messaging kernel layer **178** receives the NFS request call, a specific inter-facility message is prepared and passed to the messaging kernel layer **180** with sufficient information to identify the request and the facility that sourced the request. As illustrated, messages are exchanged between the various messaging kernel layers of the system **160**. However, the messages are in fact transferred physically via the enhanced VME bus connecting the peer-level processors upon which the specific peer-level facilities are executing. The physical to logical relationship of peer-level facilities to peer-level processors is established upon the initialization of the system **160** by providing each of the messaging kernel layers with the relevant message descriptor FIFO addresses of the peer-level processors.

In response to a message received, the messaging kernel layer **180** allocates a FS process within its multi-tasking environment to handle the communication transaction. This active FS process is used to call, carrying with it the received message contents, a local file system (LFS) server **182**. This LFS server **182** is, in essence, an unmodified instantiation **184** of the UFS. Calls, in turn, issued by this UFS **182**, ultimately intended for a device driver of a mass storage device, are directed back to the messaging kernel layer **180**. The messaging kernel layer distinguishes such device driver related functions being requested by the nature of the function call. The messaging kernel layer **180** blocks the relevant FS process while another inter-processor message is prepared and passed to a messaging kernel layer **186** of the storage facility **166**.

Since the storage facility **166** is also required to track many requests at any one time, a single manager process is used to receive messages. For throughput efficiency, this S manager process responds to FIFO interrupts, indicating that a corresponding message descriptor has just been written to the SP FIFO, and immediately initiates the SP processor operation necessary to respond to the request. Thus, the currently preferred S facility handles messages at interrupt time and not in the context of separately allocated processes. However, the messaging kernel layer **186** could alternately allocate an S worker process to service each received message request.

The message provided from the file system facility **164** includes the necessary information to specify the particular function required of the storage facility in order to satisfy the request. Within the context of the allocated active S process, the messaging kernel layer **186** calls the request corresponding function of a device driver **188**.

Depending on the availability and nature of the resource requested, the device driver **188** will, for example, direct the requested data to be retrieved from the disk array resource **24'**. As data is returned via the device driver layer **188**, the relevant S process of the messaging kernel layer **186** directs the transfer of the data into the memory resource **18'**.

In accordance with the preferred embodiments of the present invention, the substantial bulk of the memory resource **18'** is managed as an exclusive resource of the file system facility **164**. Thus, for messages requesting the transfer of data to or from the disk array **24'**, the file system facility **164** provides an appropriate shared memory address referencing a suitably allocated portion of the memory resource **18'**. Thus, as data is retrieved from the disk array **24'**, the relevant S process of the messaging kernel layer **186** will direct the transfer of data from the device driver layer **188** to the message designated location within the memory resource **18'**, as illustrated by the data path **190**.

Once the data transfer is complete, the relevant S process "returns" to the messaging kernel layer **186** and a reply message is prepared and issued by the messaging kernel layer **186** to the messaging kernel layer **180**. The relevant S process may then be deallocated by the messaging kernel layer **186**.

In response to this reply message, the messaging kernel layer **180** unblocks its relevant FS process, i.e., the process that requested the S facility data transfer. This, in turn, results in the relevant FS process executing the UFS **182** and eventually issuing a return to the messaging kernel layer **180** indicating that the requested function has been completed. In response, the messaging kernel layer **180** prepares and issues a reply message on behalf of the relevant FS process to the messaging kernel layer **178**; this message will include the shared memory address of the requested data as stored within the memory resource **18'**.

The messaging kernel layer **178** responds to the reply message from the file system facility **164** by unblocking the relevant NC process. Within that NC process's context, the messaging kernel layer **178** performs a return to the NFS server **176** with the shared memory address. The messaging kernel layer **178** transfers the data from the memory resource **18'** via the indicated data path **192** to local stored memory for use by the NFS server layer **176**. The data may then be processed through the NFS server layer **176**, IP layer **172** and the network and lower layers **170** into packets for provision onto the network **26'** and directed to the originally requesting client workstation.

Similarly, where data is received via the network layer **170** as part of an NFS write transfer, the data is buffered and processed through the NFS server layer **176**. When complete, a call by the NFS server **176** to the messaging kernel layer **178** results in the first message of an inter-facility communication transaction being issued to the file system facility **164**. The messaging kernel layer **180**, on assigning a FS process to handle the request message, replies to the relevant NC process of the messaging kernel layer **178** with an inter-facility message containing a shared memory address within the memory resource **18'**. The NFS data is then transferred from local shared memory via the data path **192** by the messaging kernel **178**. When this data transfer is complete, another inter-facility message is passed to the relevant FS process of the messaging kernel layer **180**. That process is then unblocked and processes the data transfer request through the LFS/UFS **182**. The UFS **182**, in turn, initiates, as needed, inter-facility communication transactions through the messaging kernel layer **180** to prepare for and ultimately transfer the data from the memory resource **18'** via the data path **190** and device driver **188** to the disk array resource **24'**.

The host operating system facility **168** is a substantially complete implementation of the SunOS operating system including a TCP/IP and NFS stack. A messaging kernel layer **194**, not unlike the messaging kernel layers **178**, **180**, **186** is provided to logically integrate the host facility **186** into the system **160**. The operating system kernel portion of the facility **168** includes the VFS **196** and a standard instantiation of the UFS **198**. The UFS **198** is, in turn, coupled to a mass storage device driver **200** that, in normal operation, provides for the support of UFS **198** requests by calling the messaging kernel layer **194** to issue inter-facility messages to the storage facility **166**. Thus, the storage facility **166** does not functionally differentiate between the local host facility **168** and the file system facility **164** except during the initial phase of bootup. Rather, both generally appear as unique but otherwise undifferentiated logical clients of the storage facility **166**.

Also interfaced to the VFS **196** is a conventional client instantiation of an NFS layer **202**. That is, the NFS layer **202** is oriented as a client for processing client requests directed to another file server connected through a network communications facility. These requests are handled via a TCP/UDP layer **204** of a largely conventional instantiation of the Sun NFS client stack. Connected to the layer **204** are the IP and data link layers **206**. The IP and data link layers **206** are modified to communicate directly with the messaging kernel layer **194**. Messages from the messaging kernel layer **194**, initiated in response to calls directly from the data link layer **206** are logically directed by the messaging kernel **178** directly to the data link layer **170** of a network communications facility. Similarly, calls from the IP layer **172**, recognized as not NFS requests of a local file system, are passed through the messaging kernel layers **178** and **194**

directly to the TCP/UDP layers **204**. In accordance with the preferred embodiments of the present invention, the responses by the host facility **168** in such circumstances are processed back through the entire host TCP/IP stack **214**, **204**, **206**, the messaging kernel layers **194**, **178**, and finally the data link layer **170** of an NC facility **162**.

Ancillary to the IP and data link layers **206**, a route database **208** is maintained under the control and direction of a conventional "routed" daemon application. This, and related daemons such as the "mrouid", execute in the application program layer as background processes. In order to maintain coherency between the route database **208** and the route database **174** present in the network communications facility **162**, a system call layer **212**, provided as the interface between the application program layer and the kernel functions of the host facility **168**, is modified in accordance with the present invention. The modification provides for the issuance of a message containing any update information directed to the route database **208**, from the daemons, to be provided by an inter-facility communication transaction from the messaging kernel layer **194** to the messaging kernel layer **178**. Upon receipt of such a message, the messaging kernel layer **178** directs an appropriate update to the route database **174**.

The system call layer **212** also provides for access to the TCP/UDP layers via a conventional interface layer **214** known as sockets. Low level application programs may use the system call layer **212** to directly access the data storage system by calling directly on the device driver **200**. The system call layer also interfaces with the VFS **196** for access to or by the NFS client **202** and the UFS **198**.

In addition, as provided by the preferred embodiments of the present invention, the VFS **196** also interfaces to a local file system (LFS) client layer **216**. The conventional VFS **196** implements a "mount" model for handling the logical relation between and access to multiple file systems. By this model a file system is mounted with respect to a specific file system layer that interfaces with the VFS **196**. The file system is assigned a file system ID (FSID). File operations subsequently requested of the VFS **196** with regard to a FSID identified file system will be directed to the appropriate file system.

In accordance with the present invention, the LFS. client layer **216** is utilized in the logical mounting of file systems mounted through the file system facility **164**. That is, the host facility's file oriented requests presented to the VFS **196** are routed, based on their FSID, through the LFS client layer **216** to the messaging kernel layer **194**, and, in turn, to the messaging kernel layer **180** of the file system facility **164** for servicing by the UFS **182**. The model is extended for handling network file system requests. A client workstation may then issue a mount request for a file system previously exported through the VFS **196**. The mount request is forwarded by a network communications facility **162** ultimately to a mounted daemon running in the application layer **210** of the host facility **194**. The mounted daemon response in turn provides the client with the FSID of the file system if the export is successful. Thereafter, the client's NFS file system requests received by the network communications facility **162** will be redirected, based on the FSID provided with the request, to the appropriate file system facility **164** that has mounted the requested file system.

Consequently, once a file system is mounted by the UFS **182** and exported via the network communications and host facilities **162**, **168**, file oriented NFS requests for that file system need not be passed to or processed by the host facility **168**. Rather, such NFS requests are expediently routed directly to the appropriate file system facility **164**.

The primary benefits of the present invention should now be apparent. In addition to allowing multiple, independent instantiations of the network communication, file system, storage and host facilities **162**, **164**, **166**, **168**, the immediate requirements for all NFS requests may be serviced without involving the substantial performance overhead of the VFS **196** and higher level portions of the conventional Unix operating system kernel.

Finally, another aspect of the host facility **168** is the provision for direct access to the messaging kernel layer **194** or via the system call layer **212** as appropriate, by maintenance application programs when executed within the application program layer **210**. These maintenance programs may be utilized to collect performance data from status accumulation data structures maintained by the messaging kernel layer **194** and, by utilizing corresponding inter-facility messages, the accumulated status information from status data structures in the messaging kernel layers **178**, **180** and **186**.

B. Messaging Kernel Layer Functions

The messaging kernel layers **178**, **180**, **186** and **194** each include a small, efficient multi-tasking kernel. As such, it provides only fundamental operating system kernel services. These services include simple lightweight process scheduling, message passing and memory allocation. A library of standard functions and processes provide services such as sleep( ), wakeup( ), error logging, and real time clocks in a manner substantially similar to those functions of a conventional Unix kernel.

The list below summarizes the primary function primitives of the multi-tasking kernel provided in each of the messaging kernel layers **178**, **180**, **186** and **194**.

k_register(name)	Registers the current process as a provider of a named service.
k_resolve(name)	Returns the process ID for a named service.
k_send(msg,pid)	Sends a message to a specified process and blocks until the message is returned.
k_reply(msg)	Returns a received messages to its sender.
k_null_reply(msg)	Returns an unmodified message to the sender. (Faster than k_reply(msg) because the message need not be copied back.)
k_receive()	Blocks until a message is sent to this process.

The balance of the messaging kernel layers **178**, **180**, **186** and **194** is made up of routines that presumptively implement, at least from the perspective of the balance of the facility, the functions that a given facility might request of another. These routines are promised on the function primitives provided by the multi-tasking kernel to provide the specific interface functions necessary to support the NFS stack, UFS, storage device driver, or host operating system. Since such routines do not actually perform the functions for which they are called, they may be referred to as "stub routines".

C. Inter-Facility Communication (IFC) System

Communication of information between the peer-level facilities is performed as a series of communication transactions. A transaction, defined as a request message and a reply message, occurs between a pair of messaging kernel layers, though others may "listen" in order to gather performance data or perform diagnostics. A single transaction may

be suspended, i.e., the reply message held, while the receiving messaging kernel layer initiates a separate communication transaction with another peer-level facility. Once the reply message of the second transaction is received, a properly reply to the initial communication transaction can then be made.

1. Message Descriptors and Messages

The transfer of a message between sending and receiving messaging kernel layers is, in turn, generally a two step process. The first step is for the sending messaging kernel layer to write a message descriptor to the receiving messaging kernel layer. This is accomplished by the message descriptor being written to the descriptor FIFO of the receiving peer-level processor.

The second step is for the message, as identified by the message descriptor, to be copied, either actually or implicitly, from the sending messaging kernel layer to the receiving messaging kernel layer. This copy, when actually performed, is a memory to memory copy from the shared memory space of the sending peer-level processor to that of the receiving peer-level processor. Depending on the nature of the communication transaction, the message copy will be actually performed by the sending or receiving peer-level processor, or implicitly by reference to the image of the original message kept by the messaging kernel layer that initiated a particular communication transaction.

The message identified by a message descriptor is evaluated by the receiving messaging kernel layer to determine what is to be done with the message. A message descriptor as used by a preferred embodiment of the present invention is shown in FIG. **7**. The message descriptor is, in essence, a single 32-bit word partitioned into two fields. The least significant field is used to store a descriptor modifier, while the high order 30-bit field provides a shared memory address to a message. The preferred values of the modifier field are given in Table 1.

TABLE 1

	Message Modifiers
Modifier	Meaning
0	Pointer to a message being sent
1	Pointer to a reply message
2	Pointer to message to be forwarded
3	Pointer to message acknowledging a forwarded message

For request messages that are being sent, the receiving messaging kernel layer performs the message copy. For a message that is a reply to a prior message, the sending messaging kernel layer is effectively told whether a message copy must be performed. That is, where the contents of a message have not been changed by the receiving messaging kernel layer, an implicit copy may be performed by replying with a messaging descriptor that points to the original message image within the sending messaging kernel layer's local shared memory space. Similarly for forwarding type communication transactions the receiving messaging kernel layer performs the copy. A message forwarding transaction is completed when an acknowledgement message is provided. The purpose of the acknowledgement is to notify the sending messaging kernel layer to know that it can return the reference message buffer to its free buffer pool.

The preferred block format of a message is illustrated in FIG. **8**. The message is a single data structure defined to occupy 128 bytes. The initial 32-bit word of the message encodes the message type and a unique peer-level facility identifier. The text of the message then follows with any necessary fill to reach a current maximum text limit. In the preferred embodiment of the present invention, the text length is 84 bytes. An inter-facility communication (IFC) control data block is provided, again followed by any necessary fill characters needed to complete the 128-byte long message. This IFC control data preferably includes a copy of the address of the original message, the relevant sending and receiving (destination) process identifiers associated with the current message, and any queue links required to manage the structure while in memory.

An exemplary message structure is provided in Table 2.

TABLE 2

Exemplary Message Structure

```
typedef struct m16_msg {
    K_MSGTYPE type;          /* request code */
    char      msg[84];
    vme_t     addr;          /* shared memory address of
                                the original message */
    PID       m16_sender_pid;  /* PID of last sender. */
    PID       m16_forward_pid;/* PID of last forwarder. */
    PID       m16_dest_pid;  /* PID of dest. process. */
    /*        Following value is LOCAL and need
              not be transfered. */
    struct m16_msg *m16_link;      /* message queue
                                      link */
} K_MSG;
```

This structure (K_MSG) includes the message type field (K_MSGTYPE), the message text (msg[ ]), and the IFC block (addr, **m16**_sender_pid, **m16**_sender_pid, **m16**_dest_pid, and **m16**_link). This K_MSG structure is used to encapsulate specific messages, such as exemplified by a file system facility message structure (FS_STD_T) shown in Table 3.

TABLE 3

Exemplary Specific Message Structure

```
typedef struct {
        K_MSGTYPE       type;
        long            errno;
    FC_CRED             cred;   /* Access credentials */
    FC_FH               file;   /* File handle **/
    union {
            FS_FSID     fsid;   /* For fc_get_server. */
            long        mode;   /* {READ,WRITE,EXEC} for
                                   fc_access. */
            K_PID       pid;    /* FS facility server pid */
            long        mask;   /* Mask attributes. */
    } un;
} FS_STD_T;
```

The FS_STD_T structure is overlaid onto a K_MSG structure with byte zero of both structures aligned. This composite message structure is created as part of the formatting of a message prior to being sent. Other message structures, appropriate for particular message circumstances, may be used. However, all are consistent with the use of the K_MSG message and block format described above.

2. IFC Message Generation

The determination to send a message, and the nature of the message, is determined by the peer-level facilities. In particular, when a process executing on a peer-level processor requires the support of another peer-level facility, such as to store or retrieve data or to handle some condition that it alone cannot service, the peer-level facility issues a message requesting the required function or support. This message, in accordance with the present invention, is generally initiated

in response to the same function call that the facility would make in a uniprocessor configuration of the prior art. That is, in a conventional single processor software system, execution of a desired function may be achieved by calling an appropriate routine, that, in turn, determines and calls its own service routines. This is illustrated in FIG. **9**. A function call to a routine A, illustrated by the arrow **300**, may select and call **302** a routine B. As may be necessary to carry out its function, the routine B may call **304** still further routines. Ultimately, any functions called by the routine B return to the function B which returns to the function A. The function A then itself returns with the requested function call having been completed.

In accordance with the present invention, the various messaging kernels layers have been provided to allow the independent peer-level facilities to be executed on respective processors. This is generally illustrated in FIG. **10** by the inclusion of the functions A' and B' representing the messaging kernel layers of two peer-level facilities. A function call **302** from the routine A is made to the messaging kernel A'. Although A' does not implement the specific function called, a stub routine is provided to allow the messaging kernel layer A' to implicitly identify function requested by the routine A and to receive any associated function call data; the data being needed by the routine B to actually carry out the requested function. The messaging kernel layer A' prepares a message containing the call data and sends a message descriptor **306** to the appropriate messaging kernel layer B'. Assuming that the message is initiating a new communication transaction, the messaging kernel layer B' copies the message to its own shared memory.

Based on the message type, the messaging kernel B' identifies the specific function routine B that needs to be called. Utilizing one of its own stub routines, a call containing the data transferred by the message is then made to the routine B. When routine B returns to the stub process from which it was called, the messaging kernel layer B' will prepare an appropriate reply message to the messaging kernel layer A'. The routine B return may reference data, such as the status of the returning function, that must also be transferred to the messaging kernel layer A'. This data is copied into the message before the message is copied back to the shared memory space of the A' peer-level processor. The message copy is made to the shared memory location where the original message was stored on the Al peer-level processor. Thus, the image of the original message is logically updated, yet without requiring interaction between the two messaging kernel layers to identify a destination storage location for the reply message. A "reply" message descriptor pointing to the message is then sent to the messaging kernel layer A'.

The messaging kernel layer A', upon successive evaluation of the message descriptor and the message type field of the message, is able to identify the particular process that resulted in the reply message now received. That is, the process ID as provided in the original message sent and now returned in the reply message, is read. The messaging kernel layer A' is therefore able to return with any applicable reply message data to the calling routine A in the relevant process context.

A more robust illustration of the relation between two messaging kernel layers is provided in FIG. **11**. A first messaging kernel layer **310** may, for example, represent the messaging kernel layer **178** of the network communications peer-level facility **162**. In such case, the series of stub routines A1-X include a complete NFS stack interface as well as an interface to every other function of the network communications facility that either can directly call or be called by the messaging kernel layer **178**. Consequently, each call to the messaging kernel layer is uniquely identifiable, both in type of function requested as well as the context of the process that makes the call. Where the messaging kernel layer calls a function implemented by the NFS stack of its network communications facility, a process is allocated to allow the call to operate in a unique context. Thus, the call to or by a stub routine is identifiable by the process ID, PID, of the calling or responding process, respectively.

The calling process to any of the stub routines A1-X, upon making the call, begins executing in the messaging kernel layer. This execution services the call by receiving the function call data and preparing a corresponding message. This is shown, for purposes of illustrating the logical process, as handled by the logical call format bubbles A1-X. A message buffer is allocated and attached to a message queue. Depending on the particular stub routine called, the contents of the message may contain different data defined by different specific message data structures. That is, each message is formatted by the appropriate call format bubble A1-X, using the function call data and the PID of the calling process.

The message is then logically passed to an A message state machine for sending. The A message state machine initiates a message transfer by first issuing a message descriptor identifying the location of the message and indicating, for example, that it is a new message being sent.

The destination of the message descriptor is the shared memory address of the message descriptor FIFO as present on the intended destination peer-level processor. The specific message descriptor FIFO is effectively selected based on the stub routine called and the data provided with the call. That is, for example, the messaging kernel layer **178** correlates the FSID provided with the call to the particular file system facility **164** that has mounted that particular file system. If the messaging kernel layer **178** is unable to correlate a FSID with a file system facility **164**, as a consequence of a failure to export or mount the file system, the NFS request is returned to the client with an error.

Once the message descriptor is passed to the messaging kernel layer **312** of an appropriate peer-level facility, the multi-tasking kernel of the messaging kernel layer **310** blocks the sending process until a reply message has been received. Meanwhile, the multi-tasking of the layer **310** kernel continues to handle incoming messages, initiated by reading message descriptors from its descriptor FIFO, and requests for messages to be sent based on calls received through the stub routines A1-X.

The messaging kernel layer **312** is similar to the messaging kernel layer **310**, though the implementation of the layer specifically with regard to its call format, return format, and stub routines B1-X differ from their A layer counterparts. Where, for example, the messaging kernel layer **312** is the messaging kernel layer **180** of the file system facility **164**, the stub routines B1-X match the functions of the UFS **182** and device driver **188** that may be directly called in response to a message from another facility or that may receive a function call intended for another facility. Accordingly, the preparation and handling of messages, as represented by the B message parser, call format and return format bubbles, will be tailored to the file system facility. Beyond this difference, the messaging kernel layers **310**, **312** are identical.

The B message state machine implemented by the multi-tasking kernel of the messaging kernel layer **312** receives a

message descriptor as a consequence of the peer-level processor reading the message descriptor from its message descriptor FIFO. Where the message descriptor is initiating a new message transaction, i.e., the message modifier is zero or two, the B message state machine undertakes to copy the message pointed to by the message descriptor into a newly allocated message buffer in the local shared memory of its peer-level processor. If the message modifier indicates that the message is a reply to an existing message transaction, then the B message state machine assumes that the message has already been copied to the previously allocated buffer identified by the message descriptor. Finally, if the message descriptor modifier indicates that the message pointed to by the message is to be freed, the B message state machine returns it to the B multi-tasking kernel's free message buffer pool.

Received messages are initially examined to determine their message type. This step is illustrated by the B message parser bubble. Based on message type, a corresponding data structure is selected by which the message can be properly read. The process ID of the relevant servicing destination process is also read from the message and a context switch is made. The detailed reading of the message is illustrated as a series of return format bubbles B1-X. Upon reading the message, the messaging kernel layer **312** selects a stub routine, appropriate to carry out the function requested by the received message and performs a function call through the stub routine. Also, in making the function call, the data contained by the message is formatted as appropriate for transfer to the called routine.

3. IFC Communication Transactions

FIG. **12** illustrates an exemplary series of communication transactions that are used for a network communications facility or a local host facility to obtain known data from the disk array **24** of the present invention. Similar series of communication transactions are used to read directory and other disk management data from the disk array. For clarity, the transfer of messages are referenced to time, though time is not to scale. Also for purposes of clarity, a pseudo-representation of the message structures is referenced in describing the various aspects of preparing messages.

a. LFS Read Transaction

At a time t_2 an NFS read request is received by the messaging kernel layer **178** of the network communications facility **162** from an executing (sending) process (PID=ASS). Alternately, the read request at t_2 could be from a host process issuing an equivalent LFS read request. In either case, a corresponding LFS message (message #1) is prepared (message#1.msg_type=fc_read; message#1.sender_pid=ASS; message#1.dest_pid=BSS).

The destination process (PID=BSS) is known to the messaging kernel layer **178** or **194** as the "manager" process of the file system facility that has mounted the file system identified by the FSID provided with the read request. The association of an FSID with a particular FS facility's PID is a product of the initialization of all of the messaging kernel layers.

In general, at least one "manager" process is created during initialization of each messaging kernel layer. These "manager" processes, directly or indirectly, register with a "name server manager" process (SC_NAME_SERVER) running on the host facility. Subsequently, other "manager" processes can query the "name server manager" to obtain the PID of another "manager" process. For indirect relations, the supervising "manager" process, itself registered with the "name server manager" process, can be queried for the PIDs of the "manager" processes that it supervises.

For example, a single named "file system administrator" (FC_VICE_PRES) process is utilized to supervise the potentially multiple FS facilities in the system **160**. The FC_VICE_PRES process is registered directly with the "name server manager" (SC_NAME_SERVER) process. The "manager" processes of the respective FS facilities register with the "file system administrator" (FC_VICE_PRES) process—and thus are indirectly known to the "name server manager" (SC_NAME_SERVER). The individual FS "manager" processes register with the given FSIDs of their mounted file systems. Thus, the "name server manager" (SC_NAME_SERVER) can be queried by an NC facility for the PID of the named "file system administrator" (FC_VICE_PRES). The NC facility can then query for the PID of the unnamed "manager" process that controls access to the file system identified by a FSID.

The function of a non-supervising "manager" process is to be the known destination of a message. Thus, such a "manager" process initially handles the messages received in a communication transaction. Each message is assigned to an appropriate local worker process for handling. Consequently, the various facilities need know only the PID of the "manager" process of another facility, not the PID of the worker process, in order to send a request message.

At t_3, a corresponding message descriptor (md#1vme_addr; mod=**0**), shown as a dashed arrow, is sent to the FS's messaging kernel layer **180**.

At t_4, the FS messaging kernel layer **180** copies down the message (message#1), shown as a solid arrow, for evaluation, allocates a worker process to handle the request and, in the context of the worker process, calls the requested function of its UFS **182**. If the required data is already present in the memory resource **18'**, no communication transaction with the S messaging kernel layer **186** is required, and the FS messaging kernel layer **180** continues immediately at t_{14}. However, if a disk read is required, the messaging kernel layer **180** is directed by the UFS **182** to initiate another communications transaction to request retrieval of the data by the storage facility **166**. That is, the UFS **182** calls a storage device driver stub routine of the messaging kernel layer **180**. A message (message#2), including a vector address referencing a buffer location in the memory resource **18'** (message#2.msg_type=sp_read; message#2.vme_addr=xxxxh; message#2.sender_pid=BSS; message#2.dest_pid=CSS), is prepared. At t_5, a corresponding message descriptor is sent (md#2vme_addr; mod=**0**) to the S messaging kernel layer **186**.

At t_9, the S messaging kernel layer **186** copies down the message (message#2) for evaluation, allocates a worker process to handle the request and calls the requested function of its device driver **188** in the context of the worker process. Between t_7 and t_{11}, the requested data is transferred to the message specified location (message#2.vme_addr=xxxxh) in the memory resource **18'**. When complete, the device driver returns to the calling stub routine of the S messaging kernel layer **186** with, for example, the successful (err=0) or unsuccessful (err=−1) status of the data transfer. Where there is an error, the message is updated (message#2.err=−1) and, at t_{12}, copied up to the messaging kernel layer **180** (md#2vme_addr). A reply message descriptor (md#2vme_addr; mod=1) is then sent at t_{13} to the FC messaging kernel layer **180**. However, where there is no error, a k_null_reply(msg) is used. This results in no copy of the unmodified message at t_{12}, but rather just the sending of the reply message descriptor (md#2vme_addr; mod=1) at t_{13}.

Upon processing the message descriptor and reply message (message#2), the FS messaging kernel layer **180**

unblocks and returns to the calling process of the UFS **182** (message#2.sender_pid=BSS). After completing any processing that may be required, including any additional communication transactions with the storage facility that might be required to support or complete the data transfer, the UFS **182** returns to the stub routine that earlier called the UFS **182**. The message is updated with status and the data location in the memory resource **18'** (message#1.err=0; message #2.vme_addr=xxxxh=message#1.vme_addr=xxxxh) and, at t_{14}, copied up to the messaging kernel layer **178** or **194** (md#1vme_addr). A reply message descriptor (md#1vme_addr; mod=1) is then sent at t_{15} to the messaging kernel layer of the NC or local host, as appropriate.

The messaging kernel layer **178** or **196** processes the reply message descriptor and associated message. As indicated between t_{16} and t_{19}, the messaging kernel layer **178** or **196**, in the context of the requesting process (PID=ASS), is responsible for copying the requested data from the memory resource **18'** into its peer-level processor's local shared memory. Once completed, the messaging kernel layer **178** or **196** prepares a final message (message#3) to conclude its series of communication transactions with the FS messaging kernel layer **180**. This message is the same as the first message (message#3=message#1), though updated by the FS facility as to message type (message#3.msg_type=fc_read_release) to notify the FC facility that it no longer requires the requested data space (message#3.vme_addr=xxxxh) to be held. In this manner, the FC facility can maintain its expedient, centralized control over the memory resource **18'**. A corresponding message descriptor (md#3vme_addr=md#1vme_addr; mod=0) is sent at t_{20}.

At t_{21}, the release message (message#3) is copied down by the FC messaging kernel layer **180**, and the appropriate disk buffer management function of the UFS **182** is called, within the context of a worker process of there relevant manager process (message#3.dest-Pid=BSS), to release the buffer memory (message#3.vme_addr=xxxxh). Upon completion of the UFS memory management routine, the relevant worker process returns to the stub routine of the FS messaging kernel layer **180**. The worker process and the message (message#3) are deallocated with respect to the FS facility and a reply message descriptor (md#3vme_addr; mod=1) is returned to the messaging kernel layer **178** or **196**, whichever is appropriate.

Finally, at t_{23} the messaging kernel layer **178** or **196** returns, within the context of the relevant process (PID=ASS), to its calling routine. With this return, the address of the retrieved data within the local shared memory is provided. Thus, the relevant process is able to immediately access the data as it requires.

b. LFS Write Transaction

FIG. **13** illustrates an exemplary series of communication transactions used to implement an LFS write to disk. Beginning at a time t_1, an LFS write request is received by the messaging kernel layer **178** of the network communications facility **162** from an executing process (PID=ASS) in response to an NFS write request. Alternately, the LFS write request at t_1 could be from a host process. In either case, a corresponding message (message #1) is prepared (message#1.msg_type=fc_write; message#1.sender_pid=ASS; message#1.dest_pid=BSS) and, at t_2, its message descriptor (md#1 vme_addr; mod=0) is sent to the FC messaging kernel layer **180**.

At t_3, the FC messaging kernel layer **180** copies down the message (message#1) for evaluation, allocates a worker process to handle the request by the manager process (PID=BSS), which calls the requested function of its UFS **182**. This UFS function allocates a disk buffer in the memory resource **18'** and returns a vector address (vme_addr=xxxxh) referencing the buffer to the FC messaging kernel layer **180**. The message is again updated (message#2.vme_addr=xxxxh) and copied back to the messaging kernel layer **178** or **194** (md#1vme_addr). A reply message descriptor (md#1vme_addr; mod=1) is then sent back to the messaging kernel layer **178** or **194**, at t_5.

Between t_6 and t_9, the relevant process (PID=ASS) of the NC or host facility copies data to the memory resource **18'**. When completed, the messaging kernel layer **178** or **194** is again called, at t_9, to complete the write request. A new message (message#2=message#1) is prepared, though updated with the amount of data transferred to the memory resource **18'** and message type (message#2msg_type=fc_write_release), thereby implying that the FS facility will have control over the disposition of the data. Preferable, this message utilizes the available message buffer of message#1, thereby obviating the need to allocate a new message buffer or to copy data from message#1. The message descriptor (md#2vme_addr=md#1vme_addr; mod=0) for this message is sent at t_{10}.

The message is copied down by the FC messaging kernel layer **180** and provided to a worker process by the relevant manager process (message#2.dest_pid=BSS). While a reply message descriptor might be provided back to the messaging kernel layer **178** or **194** immediately, at t_{12}, thereby releasing the local shared memory buffer, the present invention adopts the data coherency strategy of NFS by requiring the data to be written to disk before acknowledgment. Thus, upon copying down the message at t_{11}, the messaging kernel layer **180** calls the UFS **182** to write the data to the disk array **24'**. The UFS **182**, within the context of the relevant worker process, calls the messaging kernel layer **180** to initiate another communication transaction to request a write out of the data by the storage facility **166**. Thus, a storage device driver stub routine of the messaging kernel layer **180** is called. A message (message#3), including the shared memory address of a buffer location in the memory resource **18'** (message #3.msg_type=sp_write; message#2.vme_addr=xxxxh; message#2.sender_pid=BSS; message#2.dest-pid=C$S), is prepared. At t_{16}, a corresponding message descriptor is sent (md#3vme_addr; mod=0) to the S messaging kernel layer **186**.

At t_{17}, the S messaging kernel layer **186** copies down the message (message#3) for evaluation, allocates a worker process to handle the request by the manager process (PID=C$S), which calls the requested function of its device driver **188**. Between t_{18} and t_{22}, the requested data is transferred from the message specified location (message#3.vme_addr=xxxxh) of the memory resource **18'**. When complete, the device driver returns to the calling stub routine of the S messaging kernel layer **186** with, for example, the status of the data transfer (err=0). The message is updated (message#3.err=0) and, at t_{23}, copied up to the messaging kernel layer **180** (md#3vme_addr). A reply message descriptor (md#3vme_addr; mod=1) is then sent at t_{24} to the FC messaging kernel layer **180**.

Upon processing the message descriptor and reply message (message#3), the FC messaging kernel layer **180** returns to the calling process of the UFS **182** (message#3.sender_pid=BSS). After completing any UFS processing that may be required, including any additional communication transactions with the storage facility that might be required to support or complete the data transfer,

the UFS **182** returns to the messaging kernel layer **180**. At this point, the UFS **182** has completed its memory management of the memory resource **18'**. At t_{25}, the messaging kernel layer **180** sends the reply message descriptor (md#2vme__addr; mod=1) to the messaging kernel layer **178** or **196**, as appropriate, to indicate that the data has been transferred to the disk array resource **24'**.

Finally, at t_{26}, the messaging kernel layer **178** or **196** returns, within the context of the relevant worker process, to its calling routine.

c. NC/Local Host Transfer Transaction

FIG. **14** illustrates the communication transaction and delivery of data, as provided from a NC facility process (PID=ASS), to an application program executing in the application program layer of the local host facility. The packet, for example, could contain new routing information to be added to the route data base. However, since the NC facility does not perform any significant interpretation of non-NFS packets beyond identification as an IP packet, the packet is passed to the local host facility. The local host, upon recognizing the nature of the non-NFS packet, will pass it ultimately to the IP client, as identified by the packet, for interpretation. In this example, the IP client would be the "route" daemon.

Thus, the transaction begins at t_2, with the NC messaging kernel layer **178** writing a message descriptor (md#**1**.vme__addr; mod=0) to the host messaging kernel layer **194**. The referenced message (message#**1**.msg__type=nc__recv__ip__pkt; message#**1**.sender__pid=DSS; message#**1**.dest__pid=ESS) is copied down, at t_3, by the host messaging kernel layer **194**. A reply message descriptor (md#**1**.vme__addr; mod=3) is then returned to the NC messaging kernel layer **178** at t_4.

The packet is then passed, by the local host messaging kernel layer **194**, to the TCP/UDP layers **204** of the local host facility for processing and, eventually, delivery to the appropriate application program.

As shown at t_{-24}, the application program may subsequently call the host messaging kernel layer **194**, either directly or indirectly through the system call layer. This call could be, for example, issued as a consequence of the application program making a system call layer call to update the host's IP route database. As described earlier, this call has been modified to also call the host messaging kernel layer **194** to send a message to the NC facility to similarly update its IP route database. Thus, a message descriptor (md#2.vme__addr; mod=0) is sent at t_{15} to the NC messaging kernel layer **178**. The referenced message (message#**2**.msg__type=nc__add__route; message#2.sender__pid=ESS; message#1.dest__pid=DSS) is copied up, at t_{16}, by the NC messaging kernel layer **178**. The NC messaging kernel layer **178** then calls the NC facility function to update the IP route database. Finally, a reply message descriptor (md#2.vme__addr; mod=1) is returned to the local host messaging kernel layer **194** at t_{17}.

d. NC/NC Route Transfer Transaction

FIG. **15** illustrates the routing, or bridging, of a data packet two NC facility processes. The two NC processes may be executing on separate peer-level processors, or exist as two parallel processes executing within the same NC facility. The packet, for example, is intercepted at the IP layer within the context of the first process (PID=ASS). The IP layer identifies the logical NC facility that the packet is to be routed to calls the messaging kernel layer **178** to prepare an appropriate message (message#1). The data packet itself is copied to a portion of the memory resource **18'** (vme__addr=xxxxh) that is reserved for the specific NC facility; this memory is not under the control of any FS facility.

Thus, at t_2, the NC messaging kernel layer **178** writes a message descriptor (md#**1**.vme__addr; mod=0) to the second messaging kernel layer **178**. The referenced message (message#**1**.msg__type=nc__forward__ip__pkt; message#**1**.sender__pid=FSS; message#1.dest__pid=GSS; message#1.vme__addr=xxxxh; message#1.ethernet__dst__mct=xx) is copied down, at t_3, by the second NC messaging kernel layer **178**. The data packet is then copied, between t_4 and t_8, from the memory resource **18'** to the local shared memory of the second NC peer-level processor.

Since the first NC facility must manage its portion of the memory resource **18'**, the second NC messaging kernel layer **178**, at t_9, returns a reply message descriptor (md#**1**.vme__addr; mod=**1**) back to the first NC messaging kernel layer **178** at t_9. This notifies the first NC facility that it no longer requires the memory resource **18'** data space (message#**1**.vme__addr=xxxxh) to be held. In this manner, the first NC facility can maintain expedient, centralized control over its portion of the memory resource **18'**.

The packet data is then passed, by the second NC messaging kernel layer **178**, to the IP layer of its NC facility for processing.

4. Detailed Communication Transaction Messages, Syntax, and Semantics

A Notation for Communication Transactions

A terse notation for use in describing communication transactions has been developed. This notation does not directly represent the code that implements the transactions, but rather is utilized to describe them. A example and explanation of the notation is made in reference to a LFS type transaction requesting the attributes of a given file.

The communication transaction:

fc__get__attributes(FILE,ATTRIBUTES);

identifies that a message with type FC__GET__ATTRIBUTES, the expected format of the message, when sent to the FS facility, for example, is a typedef FILE, and that when the message is returned, its format is a typedef ATTRIBUTES.

A second convention makes it very clear when the FS facility, for example, returns the message in the same format that it was originally sent. The communication transaction:

get__buffer(BUFFER,***);

describes a transaction in which the NC facility, for example, sends a typedef BUFFER, and that the message is returned using the same structure.

If a facility can indicate success by returning the message unchanged (k__null__reply( )), then the format is:

free__buffer(BUFFER,*);

Sometimes, when facilities use standard structures, only some of the fields will actually have meaning. The following notation identifies meaningful fields:

```
get__buffer{ BUFFER{data__len},
             *** {data__len,data__ptr};
```

This transaction notation describes the same transaction as get__buffer above, but in more detail. The facility requests a buffer of a particular length, and the responding facility returns a pointer to the buffer along with the buffer's actual length.

a. FS Facility Communication Transactions

The communication transactions that the FS facilities of the present invention recognizes, and that the other facilities of the present invention messaging kernel layer recognize as appropriate to interact with the FS facility, are summarized in Table 4 below.

TABLE 4

Summary of FS Communication Transactions

LFS Configuration Management	
fc__find__manager	{FC__MOUNT__T,***{errno,fc__pid} }
fc__mount	{FC__MOUNT__T,***{errno,fc__pid,file} }
fc__unmount	{FC__STD__T{partition.fsid},*{errno} }
LFS Data Transfer Messages	
fc__read	{ FC__RDWR__T{un.in}, ***{errno,un.out.{bd,vattr}} }
fc__write	{ FC__RDWR__T{un.in}, ***{errno,un.out.{bd,vattr}} }
fc__readdir	{ FC__RDWR__T{un.in}, ***{errno,un.out.{bd,new__offset}} }
fc__readlink	{ FC__RDWR__T{un.in.file,un.in.cred}, ***{errno,un.out.bd} }
fc__release	{ FC__RDWR__T{un.out.bd},*{errno} }
LFS File Management Messages	
fc__null	{ K__MSG,***}
fc__null__null	{ K__MSG,*}
fc__getattr	{ FC__STD__T{cred,file,an.mask}, FC__FILE__T{errno,vattr} }
fc__setattr	{ FC__SATTR__T, FC__FILE__T{errno,vattr} }
fc__lookup	{ FC__DIROP T{cred,where}, FC__FILE__T }
fc__create	{ FC__CREATE__T, FC__FILE__T }
fc__remove	{ FC__DIROP__T{cred,where}, *{errno} }
fc__rename	{ FC__RNAMH__T, *{errno} }
fc__link	{ FC__LINK__T, *{errno} }
fc__symlink	{ FC__SYMLINK__T, *{errno} }
fc__mdir	{ FC__DIROP__T{cred,where}, *{errno} }
fc__statfs	{ FC__STATFS__T{fsid},***}
VOP, VFS and Other Miscellaneous LFS Messages	
fc__fsync	{ FC__STD__T{cred,file}, *{errno} }
fc__access	{ FC__STD__T{cred,file,mode}, *{errno} }
fc__syncfs	{ FC__STD__T{cred,fsid}, *{errno} }

The use of these communication transactions are best illustrated from the perspective of their use.

An FS facility process named FC_VICE_PRES directs the configuration of all FS facilities in the system **160**. Even with multiple instantiations of the FS facility, there is only one FC__VICE__PRES process. There are also one or more unnamed manager processes which actually handle most requests. Each file system—or disk partition—in the system **160** belongs to a particular manager; however, a manager may own more than one file system. Since managers are unnamed, would-be clients of a file system first check with FC__VICE__PRES to get the FS facility pid of the appropriate manager. Thus, the FC-VICE__PRES process does no actual work. Rather, it simply operates to direct requests to the appropriate manager.

To provide continuous service, managers must avoid blocking. Managers farm out requests that would block to a pool of unnamed file controller worker processes. These details are not visible to FS facility clients.

The significant message structures used by the FS facility are given below. For clarity, the commonly used structures are described here. An FSID (file system identifier) identifies an individual file system. An FSID is simply the UNIX device number for the disk array partition which the file system lives on. An FC__FH structure (file controller file handle) identifies individual files. It includes an FSID to identify which file system the file belongs to, along with an inode number and an inode generation to identify the file itself.

Start-up Mounting and Unmounting

Once the FC peer-level processor has booted an instantiation of the FS facility, the first FS facility to boot spawns an FC__VICE__PRES process which, in turn, creates any managers it requires, then waits for requests. Besides a few "internal" requests to coordinate the mounting and unmounting of files systems is the operation of multiple file system facilities, The only request it accepts is:

fc__find__manager (FC__MOUNT__T,***{errno,fc__pid});

The input message includes nothing but an FSID identifying the file system of interest. The successful return value is an FS facility process id which identifies the manager responsible for this file system. Having found the manager, a client facility with the appropriate permissions can request that a file system be made available for user requests (mount) or unavailable for user requests (unmount). These requests are made by the local host facility, through its VFS/LFS client interface; requests for the mounting and unmounting of file systems are not received directly from client NC facilities.

The transaction:

fc__mount (FC__MOUNT__T,***{errno,fc__pid,file});

returns the root file handle in the requested file system.

The unmount transaction:

fc__unmount (FC__STD__T{fsid}, *{errno});

returns an error code. (The * in the transaction description indicates that a k__null__reply( ) is possible, thus the caller must set errno to zero to detect a successful reply.)

Data Transfer Messages

There are four common requests that require the transfer data. These are FC__READ, FC__READDIR, FC__READLINK, and FC__WRITE. The FS facility handles these requests with a two message protocol. All four transactions are similar, and all use the FC__RDWR__T message structure for their messages.

```
typedef struct {
    void        *buf;           /* Buffer id. Valid if
                                   non-NULL. */
    vme__t      addr;           /* Pointer to data. */
    int         count;          /* Length of data. */
} FC__BUF__DESC;
#define FC__RDWR__BUFS   2
typedef struct {
    int         type;
    int         errno;
    union {
        struct {
            FC__CRED    cred; /* credentials */
            int         flags;
            FC__FH      file;
            int         offset;
            int         count;
        } in;
        struct {
            /*
             * Structure used in response to
             * fc__release message.
             */
            FC__BUF__DESC  bd[FC__RDWR__BUFS];
                        /* Buffer descriptors. */
```

-continued

```
        FC_VATTR        vattr;
    } out;
} un;
} FC_RDWR_T;
```

The FC_READ transaction is described in some detail. The three by other transactions are described by comparison.

A read data communication transactions is:

```
fc_read  { FC_RDWR_T{un.in},
           ***{errno,un.out.{bd,vattr}} );
```

As sent by a client facility, the "in" structure of the union is valid. It specifies a file, an offset and a count. The FS facility locks the buffers which contain that information; a series of message transactions with the S facility may be necessary to read the file from disk. In its reply, the FS facility uses the "out" structure to return both the attributes of the file and an array of buffer descriptors that identify the VME memory locations holding the data. A buffer descriptor is valid only if it's "buf" field is non-zero. The FS facility uses non-zero values to identify buffers, but to client facilities they have no meaning. The attributes and buffer descriptors are valid only if no error has occurred. For a read at the end of a file, there will be no error, but all buffer descriptors in the reply will have NULL "buf" fields.

After the client facility has read the data out of the buffers, it sends the same message back to the FS facility a second time. This time the transaction is:

fc_release (FC_RDWR_T{un.out.bd}, *{errno});

This fc_release request must use the same message that was returned by the fc_read request. In the reply to the fc_read, the FS facility sets the message "type" field of the message to make this work. The following pseudo-code fragment illustrates the sequence:

```
msg=(FC_RDWR_T*)k_alloc_msg( );
initialize_message;
msg=k_send(msg, fc_pid);
copy_data_from_buffers_into_local_memory;
msg=k_send(msg, fc_pid);
```

The same message, or an exact duplicate, must be returned because it contains the information the FS facility needs to free the buffers.

Although the transaction summary of Table 4 shows just one fc_release transaction, there are really four: one for each type of data transfer: fc read release, fc write release, fc_readdir_release and fc_read_link_release. Since the FS facility sets the "type" field for the second message, this makes no difference to the client facility.

If the original read transaction returned an error, or if none of the buffer descriptors were valid, then the release is optional.

The FC_WRITE transaction is identical to FC_READ, but the client facility is expected to write to the locations identified by the buffer descriptors instead of reading from them.

The FC_READDIR transaction is similar to read and write, but no file attributes are returned. Also, the specified offset is really a magic value—also sometimes referred to as a magic cookie—identifying directory entries instead of an absolute offset into the file. This matches the meaning of the offset in the analogous VFS/VOP and NFS versions of readdir. The contents of the returned buffers are "dirent" structures, as described in the conventional UNIX "getdents" system call manual page.

The FC_READLINK transaction is the simplest of the four communication transactions. It returns no file attributes and, since links are always read in their entirety, it requires no offset or count.

In all of these transactions, the requested buffers are locked during the period between the first request and the second. Client facilitics should send the fc_release message as soon as possible, because the buffer is held locked until they do, and holding the lock could slow down other client facilities when requesting the same block.

In the preferred embodiment of the present invention, the these four transactions imply conventional NFS type permission checking whenever they are received. Although conventional VFS/UFS calls do no permission checking, in NFS and the LFS of the present invention, they do. In addition, the FS facility messages also supports a "owner can always read" permission that is required for NFS.

LFS File Management Messages

The LFS communication transactions, as described below, are similar to conventional NFS call functions with the same names.

The communication transaction:

fc_null (K_MSG,***); does nothing but uses k_reply( ).

The communication transaction:

fc_null_null(K_MSG,*);

also does nothing, but uses the quicker k_null_reply( ). Both of these are intended mainly as performance tools for measuring message turnaround time.

The communication transaction:

```
fc_getattr  {FC_STD_T{cred,file,un.mask},
             FC_FILE_T{errno,vattr} };
```

gets the vnode attributes of the specified file. The mask specifies which attributes should be returned. A mask of FC_ATTR_ALL gets them all. The same structure is always used, but for un-requested values, the fields are undefined.

The communication transaction:

fc_setattr (FC_SATTR_T,FC_FILE_T{errno,vattr});

sets the attributes of the specified file. Like fc_getattr, fc_setattr uses a mask to indicate which values should be set. In addition, the special bits FC_ATTR_TOUCH_[AMC]TIME can be set to indicate that the access, modify or change time of the file should be set to the current time on the server. This allows a Unix "touch" command to work even if the times on the client and server are not well matched.

The communication transaction:

fc_lookup (FC_DIROP_T{cred,where},FC_FILE_T);

searches a directory for a specified file name, returning the file and it's attributes if it exists. The "where" field of FC_DIROP_T is an FC_DIROP structure which contains a file, a name pointer, and a name length. The name pointer contains the vme address of the name. The name may be up to 256 characters long, and must be in memory that the FS facility can read.

The communication transaction:

fc_create(FC_CREATE_T,FC_FILE_T);

creates files. The FC_CREATE_T describes what type of file to create and where. The vtype field may be used to specify any file type including directories, so mkdir is not supported. If the "FC_CREATE-EXCL" bit is set in the flag field, then fc_create will return an error if the file already exists. Otherwise, the old file will be removed before creating the new one.

The communication transaction:

fc_remove (FC_DIROP_T{cred,where},*{errno});

removes the specified name from the specified directory.

The communication transaction:

fc_rename (FC_RENAME_T,*);

changes a file from one name in one directory to a different name in a (possibly) different directory in the same file system.

The communication transaction:

fc_link (FC_LINK_T,*{errno});

links the specified file to a new name in a (possibly) new directory.

The communication transaction:

fc_symlink (FC_SYMLINK_T,*{errno});

creates the specified symlink.

The communication transaction:

fc_rmdir (FC_DIROP_T{cred,where},*{errno});

removes a directory. The arguments for fc_rmdir are like those for fc_remove.

The communication transaction:

fc_statfs (FC_STATFS_T{fsid},***);

returns file system statistics for the file system containing the specified file.

VFS/VOP LFS Support Transactions

The communication transactions described below are provided to support the VFS/VOP subroutine call interface to the LFS client layer. Most VOP calls can be provided for using the message already defined above. The remaining VOP function call support is provide by the following transactions.

The communication transactions:

fc_fsync (FC_STD_T{cred,file},*{errno});

fc_syncfs (FC_STD_T{cred,fsid}, *{errno});

ensure that all blocks for the referenced file or file system, respectively, are flushed.

The communication transaction:

fc_access(FC_STD_T{cred,file,mode},*{errno});

determines whether a given type of file access is legal for specified credentials ("cred") on the specified file. The mode value is "FC_READ_MODE", "FC_WRITE_MODE", or "FC_EXEC_MODE". If the mode is legal, the returned errno is zero.

Table 5 lists the inter-facility message types supported by the FS facility.

TABLE 5

FS Facility Message Types (K_MSGTYPE)

```
#define FC_ID ( long)( ('F'<<8) | ('C') ) << 16 )
\* External Messages *\
#define FC_FIND_MANAGER          ( 1 | FC_ID )
#define FC_MOUNT                 ( 2 | FC_ID )
#define FC_UNMOUNT               ( 3 | FC_ID )
```

TABLE 5-continued

FS Facility Message Types (K_MSGTYPE)

```
#define FC_READ                  ( 4 | FC_ID )
#define FC_WRITE                 ( 5 | FC_ID )
#define FC_READDIR               ( 6 | FC_ID )
#define FC_READLINK              ( 7 | FC_ID )
#define FC_READ_RELEASE          ( 8 | FC_ID )
#define FC_WRITE_RELEASE         ( 9 | FC_ID )
#define FC_READDIR_RELEASE       ( 10 | FC_ID )
#define FC_READLINK_RELEASE      ( 11 | FC_ID )
#define FC_NULL                  ( 12 | FC_ID )
#define FC_NULL_NULL             ( 13 | FC_ID )
#define FC_GETATTR               ( 14 | FC_ID )
#define FC_SETATTR               ( 15 | FC_ID )
#define FC_LOOKUP                ( 16 | FC_ID )
#define FC_CREATE                ( 17 | FC_ID )
#define FC_REMOVE                ( 18 | FC_ID )
#define FC_RENAME                ( 19 | FC_ID )
#define FC_LINK                  ( 20 | FC_ID )
#define FC_SYMLINK               ( 21 | FC_ID )
#define FC_RMDIR                 ( 22 | FC_ID )
#define FC_STATFS                ( 23 | FC_ID )
#define FC_FSYNC                 ( 24 | FC_ID )
#define FC_ACCESS                ( 25 | FC_ID )
#define FC_SYNCFS                ( 26 | FC_ID )
/* Internal Messages. */
#define FC_REG_PARTITION         ( 27 | FC_ID )
#define FC_UNREG_PARTITION       ( 28 | FC_ID )
```

The FS facility message structures are listed below.

```
/* Standard Structure which handles many messages. */
typedef struct {
    K_MSGTYPE  type;
    long       errno;
    FC_CRED    cred;   /* Access credentials */
    FC_FH      file;
    union {
        FC_FSID  fsid;     /* For fc_get_server. */
        long     mode;     /* (READ,WRITE,EXEC} for
                              fc_access. */
        K_PID    pid;      /* FS facility pid of
                              server. */
        long     mask;     /* Mask attributes.
                              (FC_ATTR_*). */
    } un;
} FC_STD_T;
/* Structure for fs control -- mounting, unmounting. */
typedef struct {
    K_MSGTYPE  type;
    long       errno;
    long       fc;          /* IN: Which FC to use. (i.e.
                               0, 1, . . . )*/
    long       flags;   /* IN: Mount flags. */
    FC_PARTITION  partition;   /* IN: Describes SP
                                  partition to use. */
    K_PID      fc_pid;      /* OUT: PID of manager for FS.
                               */
    FC_FH      file;        /* OUT: Root file handle of
                               file system. */
} FC_MOUNT_T;
typedef struct {
    K_MSGTYPE  type;
    FC_CRED    cred;
    FC_FH      file;
    long       mask;     /* Mask attributes.
                            (FC_ATTR_*) */
    FC_SATTR   sattr;
} FC_SATTR_T;
typedef struct {
    K_MSGTYPE  type;
    long       errno;
    FC_FH      file;
    FC_VATTR   vattr;
```

-continued

```
} FC_FILE_T;
typedef struct {
    void            *buf;
    vme_t           addr;    /* fc returned data. */
    long            count;   /* fc returned data length. */
} FC_BUF_DESC;
```

The FC_BUF_DESC structure is used in the two message data transfer protocols. A typical sequence is:

```
fc_read     { FC_RDWR_T{flags,un.in},
              FC_RDWR_T{flags,un.out} };
fc_release      { FC_RDWR_T{flags,un.out},
                  FC_RDWR_T{flags,un.out} }
```

Note that the "out" union member is the output for the first message and the input for the second.

```
#define FC_RDWR_BUFS 2
typedef struct {
    K_MSGTYPE  type;
    long       errno;
    union {
        struct {
            FC_FH       file;       /* For first
                                       message. */
            FC_CRED     cred;
            long        flags;
            long        offset;     /* User requested
                                       file offset. */
            long        count;      /* User requested
                                       count. */
        } in;
        struct {
            /* Structure used in response to
            fc_release message. */
            FC_BUF_DESC  bd[FC_RDWR_BUFS];
                                    /* Buffer
                                    descriptor. */
            FC_VATTR    vattr;      /* For
                                    responses. */
            long        new offset; /* For READDIR.
                                    */
        }
    } un;
} FC_RDWR_T;
/* #define FC_RDWR_SYNC      0x0001
/* #define FC_RDWR_NOCACHE   0x0002    /* Don't cache
                                       buffer. */
```

This structure is used in those operations that take a directory file handle and a file name within that directory, namely "lookup", "remove", and "rmdir".

```
typedef struct {
        K_MSGTYPE       type;
        long            errno;
        FC_CRED         cred;
        FC_DIROP        where;      /* File to look up
                                       or remove. */
} FC_DIROP_T;
```

Not all fields that can be set can be specified in a create, so instead of including FC_SATTR, only the values that can be set as included.

```
typedef struct {
        K_MSGTYPE       type;
        FC_CRED         cred;
        FC_DIROP        where;
        short           flag;
        short           vtype:      /* Type for new file. */
        u_short         mode:       /* Mode for new file. */
        short           major_num;  /* Major number for
                                       devices. */
        short           minor_num;  /* Minor number for
                                       devices. */
} FC_CREATE_T;
/* Values for the flag. */
#define FC_CREATE_EXCL 0x0001 /* Exclusive. */
typedef struct {
        K_MSGTYPE       type;
        long            errno;
        FC_CRED         cred;
        FC_FH           from;
        FC_DIROP        to;
} FC_RENAME_T;
typedef struct {
        K_MSGTYPE       type;
        long            errno;
        FC_CRED         cred;
        FC_FH           from;
        FC_DIROP        to;
} FC_LINK_T;
typedef struct {
        K_MSGTYPE       type;
        long            errno;
        FC_CRED         cred;
        FC_DIROP        from;       /* File to create. */
        u_short         mode;       /* File permissions. */
        vme_t           to_ptr;     /* Pointer to contents
                                    for symlink */
        long            to__len;
} FC_SYMLINK_T:
typedef struct {
        K_MSGTYPE       type;
        long            errno;
        FC_FSID         fsid;
        u_long          bsize;      /* Block size. */
        u_long          blocks;     /* Total number of
                                    blocks. */
        u_long          bfree;      /* Free blocks. */
        u_long          bavail;     /* Blocks available to
                                    non-priv users. */
        u_long          files;      /* Total number of file
                                    slots. */
        u  long         ffree;      /* Number of free file
                                    slots. */
        u_long          favail;     /* File slots available
                                    to non-priv users. */
        struct timeval stime;       /* Server's current
                                    time of day. */
} FC_STATFS_T;
#define     FC_MAXNAMLEN    255
#define     FC_MAXPATHLEN   1024
struct fc_dirent {
    u_long  d_off;      /* offset of next disk
                        directory entry */
    u_long  d_fileno;   /* file number of entry */
    u_short d_reclen;   /* length of this record */
    u_short d_namlen;   /* length of string in d_name
                        */
    char d_name[FC_ MAXNAMLEN + 1];   /* name (up to
                                      MAXNAMLEN + 1)
                                      */
};
```

b. NC Facility Communication Transactions

The communication transactions that the NC facilities of the present invention recognize, and that the other messaging kernel layers of the present invention messaging kernel layer recognize as appropriate to interact with the NC facility, are summarized in Table 6 below. The NC facility

also uses and recognizes the FS facility communication transactions described above.

TABLE 6

Summary of NC Communication Transactions

Network Interface IOCTL Messages
nc_register_dl (NC_REGISTER_DL_T,***{status})
nc_set_promis (NC_IFIOCTL_T{unit,promis}, ***{status})
nc_add_multi (NC_IFIOCTL_T{unit,mc_addr}, ***{status})
nc_del_multi (NC_IFIOCTL_T{unit,mc_addr}, ***{status})
nc_set_ifflags (NC_IFIOCTL_T{unit,flags}, ***{status})
nc_get_ifflags (NC_IFIOCTL_T{unit}, ***{status,flags})
nc set ifmetric (NC IFIOCTL T{unit,metric}, ***{status })
nc_set_ifaddr (NC_IFIOCTL_T{unit,if_addr}, ***{status})
nc_get_ifaddr (NC_IFIOCTL_T{unit}, ***{status,if_addr})
nc_get_ifstat (NC_IFSTATS_T,***)
nc_set_mneflags (NC_IFIOCTL_T{unit,flags}, ***{status})
nc_get_mneflags (NC_IFIOCTL_T{unit}, ***{status,flags})
nc_set_ip_broddr (NC_INIOCTL_T, ***)
nc_get_ip_broddr (NC_INIOCTL_T, ***)
nc_set_ip_netmask (NC_INIOCTL_T, ***)
nc_get_ip_netmask (NC_INIOCTL_T, ***)
nc_add_arp_entry (NC_ARPIOCTL_T, ***)
nc_del_arp_entry (NC_ARPIOCTL_T, ***)
nc_get_arp_entry (NC_ARPIOCTL_T, ***)
nc_add_route (NC_RTIOCTL_T, ***)
nc_del_route (NC_RTIOCTL_T, ***)
NFS Configuration Messages
nc_nfs_start (NC_NFS_START_T, *)
nc nfs export (NC NFS EXPORT T, ***{errno})
nc_nfs_unexport (NC_NFS_UNEXPORT_T, ***{errno})
nc_nfs_getstat (NC_NFS_STATS_T, ***)
Network Interface Data Messages
nc_xmit_pkt (NC_PKT_IO_T,*)
nc_recv_dl_pkt (NC_PKT_IO_T,*)
nc_recv_ip_pkt (NC_PKT_IO_T,*)
nc_recv_promis_pkt (NC_PKT_IO_T,*)
nc_forward_ip_pkt (NC_PKT_IO_T,*)
Secure Authentication Messages
ks_decrypt (KS_DECRYPT_T{netname,netnamelen,desblock}, ***{rpcstatus,ksstatus,desblock})
ks_getcred (KS_GETCRED_T{netname,netnamelen}, ***{rpcstatus,ksstatus,cred})

A network communications facility can exchange messages with the host facility, file system facility and any other network communications facility within the system **160**. The host facility will exchange messages with the network communications facility for configuring the network interfaces, managing the ARP table and IP routing table, and sending or receiving network packets. In addition, the host facility will exchange messages with the network communications facility for configuring the NFS server stack and to respond in support of a secure authentication service request. The network communications facility will exchange messages with the file system facility for file service using the external FS communication transactions discussed above. Finally, a network communication facility will exchange messages with other network communication facilities for IP packet routing.

System Call Layer Changes

The exportfs( ), unexport( ), rtrequest( ), arpioctl( ) and in_control( ) function calls in the system call layer have been modified. The exportfs( ) and unexport( ) functions are called to export new file systems and unexport an exported file system, respectively. A call to these modified functions now also initiates the appropriate NC_NFS_EXPORT or NC_NFS_UNEXPORT communication transactions to each of the network facility.

The rtrequest( ) function is called to modify the kernel routing table. A call to the modified function now also initiates an appropriate NC communication transaction (NC_ADD_ROUTE for adding a new route or NC_DEL_ROUTE for deleting an existing route) to each of the network facility.

The arpioctl( ) function is called to modify the kernel ARP table. This function has now been modified to also initiate the appropriate NC communication transaction (NC_ADD_ARP for adding a new ARP entry or NC_DEL_ARP for deleting an existing entry) to each of the network facility.

Finally, the in_control( ) function is called to configure the Internet Protocol parameters, such as setting the IP broadcast address and IP network mask to be used for a given interface. This function has been modified also initiate the appropriate NC communications transaction (NC_SET_IP_BRADDR or NC_SET_IP_NETMASK) to the appropriate network facility.

NC Facility Initialization

When a network communications facility is initialized following bootup, the following manager processes are created:

nc_nfs_vp<n>	NFS server process for processing NFS_EXPORT and NFS_UNEXPORT communication transactions from the host;
nc_dlctl<n>	Network interface control process for processing IOCTL communication transactions from the host; and
nc_dlxmit<i>	Network transmit process for processing NC_XMIT_PKT and NC_FWD_IP_PKT communication transactions.

where:

<n> is the network processor number: 0,1,2, or 3.

<i> is the network interface (LAN) number: 0,1,2,3,4,5,6, or 7.

Once initialized, the NC facilities reports the "names" of these processes to a SC_NAME_SERVER manager process, having a known default PID, started and running in the background, of the host facility. Once identified, the host facility can configure the network interfaces (each LAN connection is seen as a logical and physical network interface). The following command is typically issued by the Unix start-up script for each network interface:

ifconfig <interface name> <host name> <options> up

where:

<interface name> is the logical name being used for the interface;

<host name> is the logical host name of the referenced <interface name>.

The ifconfig utility program ultimately results in two IOCTL commands being issued to the network processor:

```
nc_set_ifflags( flags = UP + <options> );
nc_set_ifaddr( ifaddr=address_of_host-
    name(<host name>) );
```

The mapping of <host name> to address is typically specified in the "/etc/hosts" file. To start the NFS service, the following commands are typically then issued by the Unix start-up script:

nfsd <n>
exportfs -a
where:
<n> specifies number of parallel NFS server process to be started.

The nfsd utility program initiates an "nc_nfs_start" communication transaction with all network communication facilities. The "exportfs" communication transaction is used to pass the list of file systems (specified in /etc/exports) to be exported by the NFS server using the "nc-nfs-export" communication transaction.

Once the NFS service is initialized, incoming network packets address to the "NFS server UDP port" will be delivered to the NFS server of the network communications facility. It will in turn issue the necessary FS communication transactions to obtain file service. If secure authentication option is used, the NFS server will issue requests to the Authentication server daemon running on the host processor. The conventional authentication services include: mapping (ks_getcred( )) a given <network name> to Unix style credential, decrypting (ks_decrypt( )) a DES key using the public key associated with the <network name> and the secret key associated with user ID **0** (ie. with the <network name> of the local host).

Routing

Once a network communication facility is initialized properly, the IP layer of the network communication facility will perform the appropriate IP packet routing based on the local routing database table. This routing table is managed by the host facility using the "nc_add_route" and "nc_del_route" IOCTL commands. Once a route has been determined for a particular packet, the packet is dispatched to the appropriate network interface. If a packet is destined to the other network interface on the same network communication facility, it is processed locally. If a packet is destined to a network interface of another network communication facility, the packet is forwarded using the "nc_forward_ip_pkt( )" communication transaction. If a packet is destined to a conventional network interface attached to the host facility, it is forwarded to the host facility using the "nc_forward_ip_pkt( )" communication transaction.

The host facility provides the basic network front-end service for system **160**. All packets that are addressed to the system **160**, but are not addressed to the NFS stack UDP server port, are forwarded to the host facility's receive manager process using the following communication transactions:

```
nc_recv_dl_pkt ( NC_PKT_IO_T;* );
    where the packet type is not IP; and
nc_recv_ip_pkt ( NC_PKT_IO_T;* );
```

The communication transaction:

nc_recv_promis_pkt (NC_PKT_IO_T,*);

transfers packets not addressed to system **160** to the host facility when a network communication facility has been configured to receive in promiscuous mode by the host facility.

To transmit a packet, the host facility initiates a communication transaction:

nc_xmit_pkt (NC_PKT_IO_T,*);

to the appropriate network communication facility.

Finally, the host facility may monitor the messages being handled by a network communication facility by issuing the communication transaction:

nc_recv_promis_pkt (NC_PKT_IO_T,*); to the appropriate network communication facility.

Table 7 lists the inter-facility message types supported by the FS facility.

TABLE 7

NC Facility Message Types

```
#define NC_ID ( (long)( ('N'<<8) | ('C') ) << 16 )
#define NC_IOCTL_CMD_CLASS(type) (type & 0xffffff00)
/* NC "mac" ioctl's */
#define  MAC_IOCTL_CMDS                 ((1 << 4) + NC_ID)
#define  NC_REGISTER_DL                 (MAC_IOCTL_CMDS+0)
#define  NC_SET MACFLAGS                (MAC_IOCTL_CMDS+1)
#define  NC_GET  MACFLAGS               (MAC IOCTL CMDS+2)
#define  NC_GET_IFSTATS                 (MAC_IOCTL_CMDS+3)
/* BSD "if" ioctl's */
#define  DL_IOCTL_CMDS                  ((2 << 4) + NC_ID)
#define  NC_SET_PROMSIC                 (DL_IOCTL_CMDS+0)
#define  NC_ADD_MULIT                   (DL_IOCTL_CMDS+1)
#define  NC_DEL_MULIT                   (DL_IOCTL_CMDS+2)
#define  NC_SET_IFFLAGS                 (DL_IOCTL_CMDS+3)
#define  NC_GET_IFFALGS                 (DL_IOCTL_CMDS+4)
#define  NC_SET_IFMETRIC                (DL_IOCTL_CMDS+5)
#define  NC_SET_IFADDR                  (DL_IOCTL_CMDS+6)
#define  NC_GET_IFADDR                  (DL_IOCTL_CMDS+7)
/* BSD "in" ioctl's */
#define  IN_IOCTL_CMDS                  ((3 << 4) + NC_ID)
#define  NC_SET_IP_BRADDR               (IN_IOCTL_CMDS+0)
#define  NC_SET_IP_NETMASK              (IN_IOCTL_CMDS+1)
#define  NC_GET_IP_BRADDR               (IN_IOCTL_CMDS+2)
#define  NC_GET_IP_NETMASK              (IN_IOCTL_CMDS+3)
/* BSD "arp" ioctl's */
#define  ARP_IOCTL_CMDS                 ((4 << 4) + NC_ID)
#define  NC_ADD_ARP                     (ARP_IOCTL_CMDS+0)
#define  NC_DEL_ARP                     (ARP_IOCTL_CMDS+1)
#define  NC_GET_ARP                     (ARP_IOCTL_CMDS+2)
/* BSD "route" ioctl's */
#define  RT_IOCTL_CMDS                  ((5 << 4) + NC_ID)
#define  NC_ADD_ROUTE                   (RT_IOCTL_CMDS+0)
#define  NC_DEL_ROUTE                   (RT_IOCTL_CMDS+1)
/* Host/NC to NC data communication transactions. */
#define  NC_DLXMIT_MSGTYPES             ((6 << 4) + NC_ID)
#define  NC_XMIT_PKT                    (NC_DLXMIT_
                                        MSGTYPES+0)
#define  NC_FWD_IP_PKT                  (NC_DLXMIT_
                                        MSGTYPES+1)
/* Data communication transactions to host receiver
processes. */
#define  NC_DLRECV_MSGTYPES             ((7 << 4) + NC_ID)
#define  NC_RECV_DL_PKT                 (NC_DLRECV_
                                        MSGTYPES+0)
#define  NC_RECV_PROMIS_PKT             (NC_DLRECV_
                                        MSGTYPES+1)
#define  NC_RECV_IP_PKT                 (NC_DLRECV_
                                        MSGTYPES+2)
/* NFS server communication transactions */
#define  NFS_CMDS                       ((8 << 4) + NC_ID)
#define  NC_NFS_START                   (NFS_CMDS+0)
#define  NC_NFS_EXPORT                  (NFS_CMDS+1)
#define  NC_NFS_UNEXPORT                (NFS_CMDS+2)
#define  NC_NFS_GETSTAT                 (NFS_CMDS+3)
#define  NC_NFS_STOP                    (NFS_CMDS+4)
```

The NC facility message structures are listed below.

```
/*
 * exported vfs flags.
 */
#define EX_RDONLY       0x01   /* exported read only */
#define EX_RDMOSTLY     0x02   /* exported read mostly
*/
#define EXMAXADDRS 10    /* max number address list */
```

-continued

```
typedef struct {
        u_long    naddrs;          /* number of addresses */
        vme_t     addrvec;         /* pointer to array of
                                      addresses */
} NC_EXADDRLIST;
/*
 * Associated with AUTH_UNIX is an array of internet
 * addresses to check root permission.
 */
#define EXMAXROOTADDRS 10
typedef struct {
        NC_EXADDRLIST rootaddrs;
} NC_UNIXEXPORT;
/*
 * Associated with AUTH_DES is a list of network names
 * to check root permission, plus a time window to
 * check for expired credentials.
 */
#define EXMAXROOTNAMES 10
typedef struct {
        u_long names;
        vme_t rootnames;          /* names that point to
                                     netnames */
        vme_t rootnamelens;       /* lengths */
        u_int window;
} NC_DESEXPORT;
typedef struct {
        long val[2];              /* file system id type */
} fsid_t;
/* File identifier. Should be unique per filesystem
   on a single machine.
 */
#define MAXFIDSZ 16
struct fid {
        u_short  fid_len; /* length of data in bytes */
        char     fid_data[MAXFIDSZ]; /* data */
};
/*********************************************************
 * NFS Server Communication transaction Structures.
 *********************************************************/
typedef struct {
        K_MSGTYPE  m_type;
        int        nservers;      /* number of servers to start
                                     up */
} NC_NFS_START_T;
typedef struct {
        K_MSGTYPE  m_type;
        long       errno;         /* error returned */
        fsid_t     fsid;          /* FSID for directory being
                                     exported */
        struct fid fid;           /* FID for directory being
                                     exported */
        long       flags;         /* flags */
        u_short    anon;          /* uid for unauthenticated
                                     requests */
        long       auth;          /* switch for authentication
                                     type */
        union {
            NC_UNIXEXPORT exunix;  /* case AUTH_UNIX */
            NC_DESEXPORT exdes;    /* case AUTH_DES */
        } ua;
        NC_EXADDRLIST writeaddrs;
} NC_NFS_EXPORT_T;
typedef struct {
        K_MSGTYPE  m_type;
        long       errno;         /* error returned */
        fsid_t     fsid;          /* of directory being
                                     unexported */
        struct fid fid;           /* FID for directory being
                                     unexported */
} NC_NFS_UNEXPORT_T;
/*
 * Return server statistics.
 */
typedef struct {
        int   ncalls;     /* Out - total RPC calls */
        int   nbadcalls;  /* Out - bad RPC calls */
        int   reslimit;
        int   nbadlen;
        int   nxdrcall;
        int   ncalls;     /* Out - total NFS calls */
        int   nbadcalls;  /* - calls that failed */
        int   reqs[32];   /* - calls for each request */
} NC_NFS_STATS_T;
/*
 * Network Interface IOCTL communication transaction
structures
 *-------------------------------------------------------*/
typedef struct {
        K_MSGTYPE  m_type;
        short      status;        /* output */
        char       unit;          /* Only used with IF, MAC and
                                     IN commands. */
        char       pad;
        K_PID      receiver_pid;
        short      mem_xfer_mode:     /* 0-normal, 1-VME
                                         block, 2-AEP */
        long       recv_mem_size;     /* I */
        long       recv_mem_start_addr;  /* I */
        ETHADDR    intf_addr;         /* O: address of
                                         interface */
} NC_REGISTER_DL_T;
typedef struct {
        K_MSGTYPE  m_type;
        short      status;        /* output */
        char       unit;          /* Only used with IF, MAC and
                                     IN commands. */
        char   pad;
        union {
            long       promis;    /* I */
            ETHADDR    mc_addr;   /* I: add and delete */
            short      flags;     /* I: set flag; O: get
                                     flag */
            long       metric;    /* I */
            struct sockaddr if_addr;
                                  /* I */
        } un;
} NC_IFIOCTL_T;
typedef struct {
        K_MSGTYPE  m_type;
        short      status;        /* output */
        char       unit;          /* Only used with IF, MAC and
                                     IN commands. */
        char   pad;
        struct if_stats {
        long if_ipackets;         /* packets received */
        long if_ibytes;           /* bytes received */
        long if_ierrors;          /* input errors */
        long if_opackets;         /* packets sent */
        long if_obytes;           /* bytes sent */
        long if_oerrors;          /* output errors */
        long if_collisions;       /* CSMA collisions */
        } if_stats;
} NC_IFSTATS_T;
typedef struct {
        K_MSGTYPE  m_type;
        short      status;        /* output */
        char       unit;          /* Only used with IF, MAC and
                                     IN commands. */
        char       pad;
        union {
            struct in_addr br_addr;   /* I */
            struct in_addr net_mask;  /* I */
        } un;
} NC_INIOCTL_T;
typedef struct {
        K_MSGTYPE  m_type;
        short      status;        /* output */
        char       unit;          /* only used with IF, MAC and
                                     IN commands. */
        char       pad;
        struct arpreq  arp_req;
} NC_ARPIOCTL_T;
typedef struct {
        K_MSGTYPE  m_type;
        short      status;        /* output */
        char       unit;          /* Only used with IF, MAC and
                                     IN commands. */
        char       pad;
```

-continued

```
    struct rtentry  route_req;
} NC_RTIOCTL_T;
/*_____________________________________________________
 * Network Interface Data Communication transaction
Structure
 *_____________________________________________________*/
typedef struct {
    long      len;
    caddr_t   address;
} PKT_DATA_BUFFER;
#define   MAX_DL_BUFFRAG          4
#define   VME_XFER_MODE_NORMAL    0
#define   VME_XFER_BLOCK          1
#define   VME_XFER_AEP            2     /* enhanced
                                          protocol */

typedef struct ether_xmit {
    K_MSGTYPE  m_type;
    char    src_net;   /* Source of packet. */
    char    dst_net;   /* Destination of packet. */
    char    vme_xfer_mode; /* What transfer mode can be
                              used to access data in
                              buffist. */
    char    pad1;
    short   pktlen;   /* Total packet length. */
    short   pad2;
    PKT_DATA_BUFFER pkt_buflist[MAX_DL_BUFFRAG+1];
} NC_PKT_IO_T;
/*******************************************************
 * Secure Authentication Server Communication
transactions
 *******************************************************/
/*
 * Name under which the key server registers.
 */
#define   KEYSERV_NAME  "KEYSERV"
/* Key server message types. */
#define   KS_DECRYPT    69
#define   KS_GETCRED    137
typedef struct {
    K_MSGTYPE  type;
    u_long     rpcstatus;     /* RPC status */
    u_long     ksstatus;      /* key server reply
                                 status */
    vme_t      netname;       /* netname */
    long       netnamelen;    /* length of netname */
    des_block  desblock;      /* DES block in and out
*/
} KS_DECRYPT_T;
typedef struct {
    K_MSGTYPE  type;
    u_long     rpcstatus;     /* RPC status */
    u_long     ksstatus;      /* key server reply
                                 status */
    vme_t      netname;       /* netname */
    long       netnamelen;    /* length of netname */
    unixcred   cred;          /* credentials returned */
} KS_GETCRED_T;
```

c. Host Facility Communication Transactions

The communication transactions that the host facility of the present invention recognizes and provides are summarized in Table 8 below. These transactions are used to support the initialization and ongoing coordinated operation of the system **160**.

TABLE 8

Host Facility Message Types	
sc_register_fifo	(SC_REGISTER_FIFO_T,***);
sc_get_sys_config	(SC_GET_SYS_CONFIG_T,***);
sc_register_name	(SC_REGISTER_NAME_T,***);
sc_init_complete	(SC_INIT_COMPLETE_T,***);
sc_resolve_name	(SC_RESOLVE_NAME_T,***);

TABLE 8-continued

Host Facility Message Types	
sc_resolve_fifo	(SC_RESOLVE_FIFO_T,***);
sc_time_register	(SC_TIME_REGISTER_T,***);
sc_real_time	(SC_REAL_TIME_T,***);
sc_err_log_msg	(SC_ERR_LOG_MSG_T,***);
sc_err_log_msg2	(SC_ERR_LOG_MSG2,***);

Name Service

The name server daemon ("named") is the Unix host facility process that boots the system and understands all of the facility services that are present in the system. That is, each facility provides at least one service. In order for any facility to utilize a service of another, the name of that service must be published by way of registering the name with the name server daemon. A name is an ascii string that represents a service. When the name is registered, the relevant servicing process PID is also provided. Whenever the name server daemon is thereafter queried to resolve a service name, the name server daemon will respond with the relevant process PID if the named service is available. This one level of indirection relieves the need to otherwise establish fixed process IDs for all of the possible services. Rather, the multi-tasking kernels of the messaging kernel layers are allowed to establish a PID of their own choosing to each of the named services that they may register.

The communication transaction:

sc_register_fifo (SC_REGISTER_FIFO_T,***);

is directed to the named daemon of the host facility to provide notice that the issuing NC, FS, or S facility has been started. This transaction also identifies the name of the facility, as opposed to the name of a service, of the facility that is registering, its unique facility ID (VME slot ID) and the shared memory address of its message descriptor FIFO.

The communication transaction:

sc_get_sys_config (SC_GET_SYS_CONFIG_T, ***);

is used by a booting facility to obtain configuration information about the rest of the system **160** from the name server daemon. The reply message identifies all facilities that have been registered with the name server daemon.

The communication transaction:

sc_init_complete (SC_INIT_COMPLETE_T,***);

is sent to the name server daemon upon completion of its initialization inclusive of handling the reply message to its sc_get_sys_config transaction. When the name server daemon returns a reply message, the facility is cleared to begin normal operation.

The communication transaction:

sc register name (SC REGISTER NAME T,***);

is used to correlate a known name for a service with the particular PID of a facility that provides the service. The names of the typical services provided in the preferred embodiment of the present invention are listed in Table 9.

TABLE 9

Named Facility Services
Host Facility Resident
SC_NAMF_SERVER - the "Name server" daemon - executes on the host peer-level processor, or primary host processor if there is more than one host facility present in the system.

TABLE 9-continued

Named Facility Services
Provides the system wide name service. Operates also to collect and distribute information as to the configuration, both physical (the total number of NCs present in the system and the VME slot number of each) and logical (what system services are available).
SC_ERRD - the "ERRD" daemon - executes on the host peer-level processor, or primary host processor if there is more than one host facility present in the system. Injects an error message into the UNIX syslogd system. This results in the error message being printed on the system console and, typically, logged it in an error file.
SC_TIMED - the "TIMED" daemon - executes on the host peer-level processor, or primary host processor if there is more than one host facility present in the system. Returns the current system time. Can also be instructed to give notification of any subsequent time changes.
SC_KEYSERV - executes on the host peer-level processor, or primary host processor if there is more than one host facility present in the system. When NFS runs in secure (DES encryption) mode, it provides access to the conventional Unix daemon that, in turn, provides access to keys which authenticate users.
FS Facility Resident
FC_VICE_PRES - executes on the FC peer-level processor, or primary FC processor if there is more than one such facility present in the system. Coordinates the operation of multiple FS facilities by servicing all requests to identify the PID of the unnamed manager process that controls access to a FSID. At least one unnamed manager process runs in each FS facility.
FC_STATMAN# - executes in a respective FC facility (#). Functions as a "statistics manager" process on the FC facility to collect and allow other facilities to request a report of current statistics, such as the number of messages received.
S Facility Resident
S_MANAGER# - executes the respective S facility (#). All low-level disk requests for the disk array coupled to the storage processor (#) are directed to this manager process. Unnamed worker processes are allocated, as necessary to actually carry out the request.
S_STATMAN# - executes in a respective S facility (#). Functions as a "statistics manager" process on the S facility to collect and allow other facilities to request a report of current statistics.
NC Facility Resident
NC_NFS_VP# - executes in a respective NC facility (#). Controls the operation of NFS for its respective NC facility. Accepts messages from the host facility for starting and stoping NFS and for controlling the export and unexport of selected file systems.
NC_DLCTRL# - executes in a respective NC facility (#). Functions as the Data Link controller for its NC facility (#). Accepts ioctl commands for a local message specified data link and allocates a worker process, as necessary, to carry out the message request.
NC_DLXMIT# - executes in a respective NC facility (#). Functions as the Data Link transmitter for its NC facility (#). Accepts transmit commands for a local message specified data link and allocates a worker process, as necessary, to carry out the message request.
NC_STATMAN# - executes in a respective NC facility (#). Functions as a "statistics manager" process on the NC facility to collect and allow other facilities to request a report of current statistics.

The communication transaction:

sc_resolve_name (SC_RESOLVE_NAME_T,***); is used by the messaging kernel layer of a facility to identify the relevant process PID of a service provided by another facility. The reply message, when returned by the name server daemon, provides the "resolved" process ID or zero if the named service is not supported.

The communication transaction:

sc_resolve_fifo (SC_RESOLVE_FIFO_T,***);

is issued by a facility to the name server daemon the first time the facility needs to communicate with each of the other facilities. The reply message provided by the name server daemon identifies the shared memory address of the message descriptor FIFO that corresponds to the named service.

Time Service

The time server daemon ("timed") provides system wide timer services for all facilities.

The communication transaction:

sc_time_register (SC_TIME_REGISTER_T,***);

is issued by a facility to the timed daemon to determine the system time and to request periodic time synchronization messages. The reply message returns the current time.

The communication transaction:

sc_real_time (SC_REAL_TIME_T,***);

is issued by the time server daemon to provide "periodic" time synchronization messages containing the current time. These transactions are directed to the requesting process, based the "client_pid" in the originally requesting message. The period of the transactions is a function of a default time period, typically on the order of several minutes, or whenever the system time is manually changed.

Error Logger Service

The error server daemon ("errd") provides a convenient service to send error messages to the system console for all facilities.

The communication transaction:

sc_err_log_msg (SC_ERR_LOG_MSG_T,***);

prints the string that is provided in the send message, while the transaction:

sc_err_log_msg2 (SC_ERR_LOG_MSG2,***);

provides a message and an "error id" that specifies a print format specification stored in an "errd message format" file. This format file may specify the error message format in multiple languages.

```
/*********************************************
 * Structures and Constants for the SC_NAMED process.
 *********************************************/
/*
 * Board types.
 */
#define   BT_NONE          0
#define   BT_UNIX          1    /* Host Processor */
#define   BT_PSA           2    /* storage Processor */
#define   BT_FC            3    /* File Controller. */
#define   BT_NC            4    /* Network Controller. */
#define   BT_PLESSEY       5    /* Test Environment */
#define   BT_TRACE_ANAL    6    /* Message Trace
                                  Analyzer. */
#define   BT_MEM           7    /* memory board. */
/*
 * Slot descriptor.
 */
typedef struct {
    short board_type;
    short slot_id;
} SLOT_DESC_T;
/*************************************
 * SC_NAMED: Types and structure.
 *************************************/
#define SC_MSG_GROUP        ( (long)( ('S'<<8) | ('C') )
<< 16 )
#define SC_REGISTER_FIFO    (1 | SC_MSG_GROUP )
#define SC_RESOLVE_FIFO     (2 | SC_MSG_GROUP )
#define SC_REGISTER_NAME    (3 | SC_MSG_GROUP )
#define SC_RESOLVE_NAME     (4 | SC_MSG_GROUP )
#define SC_DELAY            (5 | SC_MSG_GROUP )
#define SC_GET_SYS_CONFIG   (6 | SC_MSG_GROUP )
#define SC_INIT_COMPLETE    (7 | SC_MSG_GROUP )
#define K_MAX_NAME_LEN      32   /* Maximum process
                                    name length. */

typedef struct {
    K_MSGTYPE           type;
    short               my_slot_id;
    short               sender_slot_id;
    char                name[K_MAX_NAME_LEN];
    M16_FIFO_DESC       fifo_desc;
    short               flags;                  /* flags defined
below */
} SC_REGISTER_FIFO_T;
/*
 * SC_REGISTER_FIFO_T flags:
 */
#define NO_CM_ACCESS        1    /*   can't access common
memory */
typedef struct {
    K_MSGTYPE           type;
    short               my_slot_id;
    short               dest_slot_id;
    M16_FIFO_DESC       fifo_desc;      /* 0 => not found */
} SC_RESOLVE_FIFO_T;
typedef struct {
    K_MSGTYPE           type;
    K_PID               pid;
    char                name[K_MAX_NAME_LEN];
} SC_REGISTER_NAME_T;
typedef struct {
    K_MSGTYPE           type;
    K_PID               pid;            /* 0=> not found */
    char                name[K_MAX_NAME_LEN];    /*   input
*/
} SC_RESOLVE_NAME_T;
typedef struct {
    K_MSGTYPE           type;
    SLOT_DESC_T         config[M16_MAX_VSLOTS];
} SC_GET_SYS_CONFIG_T;
typedef struct {
    K_MSGTYPE           type;
    short               my_slot_id;
} SC_INIT_COMPLETE_T;
/*********************************************
 * SC_TIMED: Types and structures.
 *********************************************/
#define   SC_TIMED_REGISTER    ( 101 | SC_MSG_GROUP )
```

-continued

```
#define SC_REAL_TIME           ( 102 | SC_MSG_GROUP )
typedef struct {
    K_MSGTYPE type;
    K_PID client_pid;
    long max_update_period; /* in seconds. */
    /* output */
    long seconds;       /* seconds since Jan. 1, 1970
*/
    long micro_seconds;     /* and micro seconds. */
} SC_TIMED_REGISTER_T;
typedef struct {
    K_MSGTYPE type;
    long seconds;           /* seconds since Jan. 1, 1970
*/
    long micro_seconds;     /* and micro seconds. */
} SC_REAL_TIME_T;
/*************************************
 * SC_ERRD: Types and Structures.
 *************************************/
/*
 * SC_ERRD message structures.
 *
 * Error log usage notes:
 *      - Must include "syslog.h"
 *      - Priority levels are:
 *          LOG_EMERG       system is unusable
 *          LOG_ALERT       action must be taken
                            immediately
 *          LOG_CRIT        critical conditions
 *          LOG_ERR         error conditions
 *          LOG_WARNING     warning conditions
 *          LOG_NOTICE      normal condition
 *          LOG_INFO        informational
 *          LOG_DEBUG       debug-level messages
 */
#define SC_ERR_LOG_MSG      (301 | SC_MSG_GROUP )
#define SC_ERR_LOG_MSG2     (302 | SC_MSG_GROUP )
#define ERR_LOG_MSG_LEN (K_MSGTYPE_SIZE -
                        sizeof(K_MSGTYPE)
                        - sizeof(short))
typedef struct {
    K_MSGTYPE type;             /* SC_ERR_LOG_MSG */
    short priority_level;
    char msg[ERR_LOG_MSG_LEN];  /* message */
} SC_ERR_LOG_MSG_T;
typedef struct {
    K_MSGTYPE type;     /* SC_ERR_LOG_MSG */
    short id;           /* Message id */
    short fill1;        /* Unused. */
    union {
      char c[80];       /* constants. */
      short s[40];
      long l[20];
    } data;
} SC_ERR_LOG_MSG2_T;
```

d. S Facility Communication Transactions

The communication transactions that the S facilities of the present invention recognize, and that the other messaging kernel layers of the present invention recognize as appropriate to interact with the S facility, are summarized in Table 10 below.

TABLE 10

Summary of S Communication Transactions

sp_noop_msg	(SP_MSG_***);
sp_send_config	(SEND_CONFIG_MSG_***);
sp_receive_config	(RECEIVE_CONFIG_MSG_***);
sp_r/w_sector	(SP_RDWR_MSG_***);
sp_r/w_cache_pg	(SP_RDWR_MSG_***);
sp_ioctl_req	(SP_IOCTL_MSG_***);
sp_start_stop_msp	(SP_IOCTL_MSG_***);

TABLE 10-continued

Summary of S Communication Transactions	
sp__inquiry__msg	(SP__MSG,***);
sp__send__message__buffer__msg	(SP__MSG,***);
sp__set__sp__interrupt__msg	(SP__MSG,***);

The S facility generally only responds to communication transactions initiated by other facilities. However, a few communication transactions are initiated by the S facility at boot up as part of the initial system configuration process.

Each S facility message utilizes the same block message structure of the FS and NC facility messages. The first word provides a message type identifier. A second word is generally defined to return a completion status. Together, these words are defined by a SP__HEADER structure:

```
typedef {
        char reserved;              /* byte 0 */
        char msg__code;             /* byte 1 */
        char msg__modifier;         /* byte 2 */
        char memory__type;          /* byte 3 */
        char complete__status;      /* byte 4 */
        char bad__drive;            /* byte 5 */
        char sense__key;            /* byte 6 */
        char sense__code;           /* byte 7 */
} SP__HEADER;
```

The reserved byte will be used by the other facilities to identify a S facility message. Msg__code and msg__modifier specify the S facility functions to be performed. Memory__type specifies the type of VME memory where data transfer takes place. The S facility uses this byte to determine the VMEbus protocols to be used for data transfer. Memory__type is defined as:

03—Primary Memory, Enhanced Block Transfer

01—Local Shared Memory, Block transfer

00—Others, Non-block transfer

The completion status word is used by the S facility to return message completion status. The status word is not written by the S facility if a message is completed without error. One should zero out the completion status of a message before sending it to the S facility. When a reply is received, one examines the completion status word to differentiate a k__reply from a k__null__reply.

The bad__drive value specifies any erroneous disk drive encountered. The higher order 4 bits specify the drive SCSI ID (hence, the drive set); the lower order 4 bits specify the S facility SCSI port number. The sense key and sense code are conventional SCSI error identification data from the SCSI drive.

The currently defined S facility functions, and identifying msg__code bytes are listed in Table 11.

TABLE 11

S Facility Message Types
01 -- No Op
02 -- Send Configuration Data
03 -- Receive Configuration Data
04 -- S facility I/O Initialization
05 -- Read and Write Sectors
06 -- Read and Write Cache Pages
07 -- IOCTL Operation

TABLE 11-continued

S Facility Message Types
08 -- Dump S facility Local RAM
09 -- Start/Stop A SCSI Drive
0A -- not used
0B -- not used
0C -- Inquiry
0D -- not used
0E -- Read Message Log Buffer
0F -- Set S facility Interrupt

The message completion status word (byte 4–7 of a message) is defined as:

Byte **00**—completion status

01—SCSI device ID and S facility SCSI port number

02—SCSI sense key

03—SCSI sense code

The completion status byte values are defined below:

00—Completed without error

01—Reserved

02—SCSI Status Error on IOCTL Message

03—Reserved

04—An inquired message is waiting to be executed

05—An inquired message is not found

06—VME data transfer error

07—Reserved

08—Invalid message parameter

09—Invalid data transfer count or VME data address

0A—S facility configuration data not available

0B 13 Write protect or drive fault

0C—Drive off-line

0D—Correctable data check

0E—Permanent drive error or SCSI interface error

0F—Unrecovered data check

After receiving a message, the S facility copies the contents into its memory. After a message's function is completed, a k__reply or k__null__reply is used to inform the message sender. K__null__reply is used when the processing is completed without error; k__reply is used when the processing is completed with error. When k__reply is used, a non-zero completion status word is written back to the original message. Therefore, when a reply is received, a message sender checks the status word to determine how a message is completed. When k__null__reply is used, the original message is not updated. The S facility simply acknowledges the normal completion of a message.

If a message is not directed to a disk drive, it is executed immediately. Disk I/O messages are sorted and queued in disk arm elevator queues. Note, the INQUIRY message returns either **04** or **05** status and uses k__reply only.

No Op

The input parameters for this message are defined as:

sp__noop__msg (SP__MSG,***);

The only parameter needed for this message is the message header. The purpose for this message is to test the communication path between the S facility and a message sender. A k__null__reply is always used.

Send Configuration Data

The input parameters for this operation are defined as:

sp__send__config (SEND__CONFIG__MSG,***);

This message is used to inform the S facility about the operating parameters. It provides a pointer pointing to a

configuration data structure. The S facility fetches the configuration data to initialize its local RAM. The configuration data is also written to a reserved sector on each SCSI disk such that they can be read back when the S facility is powered up. Hence, it is not necessary to send this message each time the S facility is powered up.

In the configuration data structure, vme_bus_request_level specifies the S facility data transfer request level on the VME bus. The access_mode specifies if the S facility should run as independent SCSI drives or as a single logical drive. In the latter case, number_of_disks should be same as number_of_banks because all nine drives in a bank are grouped into a single logical disk.

Total_sector is the disk capacity of the attached SCSI disks. Total capacity of a disk bank is this number multiplying the number_of_disks. When additional disk banks are available, they could have sizes different from the first bank. Hence, total_sector is a three_entry array. Stripe_size is meaningful only when the S facility is running as a single logical disk storage subsystem. Different stripe sizes can be used for different drive banks. Finally, online_drive_bit_map shows the drives that were online at the last reset. Bit **5** of online_drive_bit-map[**1**] being set indicates drive **5** of bank 1 is online. Total_sector and online_drive_bit_map could not and should not be specified by a user.

The configuration data are written to the disks in a S facility reserved sector, which is read at every S facility reset and power up. When the configuration data are changed, one must reformat the S facility (erase the old file systems). When this message is completed, a k_reply or k_null_reply is returned.

Receive Configuration Data

The input parameters for this operation are defined as:

sp_receive_config (RECEIVE_CONFIG_MSG,***);

This message requests the S facility to return configuration data to a message sender. Vme_pointer specifies a VME memory location for storing the configuration data. The same configuration data structure specified n the last section will be returned.

Read and Write Sectors

The input parameters for this operation are defined as:

sp_r/w_sector (SP_RDWR_MSG,***);

Unlike most S facility messages, which are processed immediately, this message is first sorted and queued. Up to 200 messages can be sent to the S facility at one time. Up to thirty messages are executed on thirty SCSI drives simultaneously. The messages are sorted by their sector addresses. Hence, they are not served by the order of their arrivals.

There are two possible functions specified by this message:

msg_mod=**00**—Sector Read-**01**—Sector Write

Scsi_id specifies the drive set number. Disk_number specifies which SCSI port to be used. Sector-count specifies the number of disk sectors to be transferred. For a sector_read message, erase_sector_count specifies the number of sectors in the VME memory to be padded with zeros (each sector is 512 bytes). For a sector_write message, erase_sector_count specifies the number of sectors on the disk to be written with zeros (hence, erased). To prevent sectors from being erased inadvertently, a sector_write message can only specify one of the two counters to be non-zero, but not both. Sector_address specifies the disk sector where read or write operation starts. Vme address specifies a starting VME memory location where data transfer takes place.

There are three drive elevator queues maintained by the S facility for each SCSI port (or one for each disk drive). The messages are inserted in the queue sorted by their sector addresses, and are executed by their orders in the queue. The S facility moves back and forth among queue entries like an elevator. This is done to minimize the disk arm movements. Separate queues for separate disk drives. These queues are processed currently because the SCSI drive disconnects from the bus whenever there is no data or command transfer activities on the bus.

If no error conditions are detected from the SCSI drive(s), this message is completed normally. When data check is found and the S facility is running as a single logical disks, recovery actions using redundant data are started automatically. When a drive is down and the S facility is running as a single logical disk, recovery actions similar to data check recovery will take place. Other drive errors will be reported by a corresponding status code value.

K_reply or K_null_reply is used to report the completion of this message.

Read/Write Cache Pages

The input parameters for this operation are defined as:

sp_r/w_cache_pg (SP_RDWR_MSG,***);

This message is similar to Read and Write Sectors, except multiple vme_addresses are provided for transferring disk data to and from disk sectors. Each vme_address points to a memory cache page, whose size is specified by cache_page_size. When reading, data are scattered to different cache pages; when writing, data are gathered from different cache pages (hence, it is referred to as scatter_gather function).

There are two possible functions specified by this message;

msg_mod=**00**—Cache Page Read=**01**—Cache Page Write

Scsi_id, disk_number, sector_count, and sector_address are described in Read and Write Sector message. Both sector_address and sector_count must be divisible by cache_page_size. Furthermore, sector_count must be less than 160 (or 10 cache pages). Cache_page_size specifies the number of sectors for each cache page. Cache pages are read or written sequentially on the drive(s). Each page has its own VME memory address. Up to 10 vme_addresses are specified. Note, the limit of 10 is set due to the size of a S facility message. Like the sector read/write message, this message is also inserted in a drive elevator queue first.

If no error conditions are detected from the SCSI drive(s), this message is completed normally. When an error is detected, a data recover action is started. When there is a permanent drive error that prevents error recovery action from continuing, an error status code is reported as completion.

K_reply or K_null_reply is used to report the completion of this message.

IOCTL Request

The input parameters for this operation are defined as:

sp_ioctl_req (SP_IOCTL_MSG,***),

This message is used to address directly any SCSI disk or peripheral attached to a SCSI port. Multiple messages can be sent at the same time. They are served in the order of first come first serve. No firmware error recovery action is attempted by the S facility.

Scsi_id, scsi_port, and scsi_lun_address identify uniquely one attached SCSI peripheral device. Command_length and data_length specify the lengths of command and data transfers respectively. Data_buffer_address points to a VME memory location for data transfer. The command_

bytes are actual SCSI command data to be sent to the addressed SCSI peripheral device. Note, the data length must be multiples of 4 because the S facility always transfers 4 bytes at a time. Sense_length and sense_addr specify size and address of a piece of VME memory where device sense data can be stored in case of check status is received. These messages are served by the order of their arrivals.

When this message is terminated with drive error, a corresponding status code is returned. K_reply and k_null_reply are used to report the completion of this message.

Start/Stop SCSI Drive

The input parameters for this operation are defined as:

sp_start_stop_msp (SP_IOCTL_MSG,***);

This message is used to fence off any message to a specified drive. It should be sent only when there is no outstanding message on the specified drive. Once a drive is fenced off, a message directed to the drive will receive a corresponding error status back.

When the S facility is running as a single logical disk, this message is used to place a SCSI disk drive in or out of service. Once a drive is stopped, all operations to this drive will be fenced off. In such case, when the stopped drive is accessed, recovery actions are started automatically. When a drive is restarted, the data on the drive is automatically reconfigured. The reconfiguration is performed while the system is online by invoking recovery actions when the reconfigured drive is accessed.

When a drive is reconfigured, the drive configuration sector is updated to indicate that the drive is now a part of a drive set.

Message Inquiry

The input parameters for this message are defined as:

sp_inquiry_msg (SP_MSG,***);

This message requests the S facility to return the status of a message that was sent earlier. A k_reply is always used. The status of the message, if available in the S facility buffers, is returned in the completion status word.

This message is used to verify if a previous message was received by the S facility. If not, the message is lost. A lost message should be resent. Message could be lost due to a local board reset. However, a message should, in general, not be lost. If messages are lost often, the S facility should be considered as broken and fenced off.

Read Message Log

The input parameters for this message are defined as:

sp_read_message_buffer_msg (SP_MSG,***);

The S facility keeps a message buffer which contains the last 200 messages. Data buffer specifies a piece of VME memory in which the messages are sent. Number_of_message should not exceed 200. Each message is 128 bytes long as defined at the beginning of this Section. An application program must allocate a buffer big enough to accommodate all returned messages.

Normally this message is sent when there is no active messages. Otherwise, it is very difficult to determine how many used messages are in the S facility message buffer. For example if there are 200 active messages, there will be no used ones in the message buffer. Where there are less than requested messages in the message buffer, 128 bytes of zeros are transmitted for each shortage. K_reply and k_null_reply are used for the completion of this message.

SP_Interrupt

The input parameters for this message are defined as:

sp_set_sp_interrupt_msg (SP_MSG,***);

This message tells the S facility to pass control to an on-board debug monitor, as present in the SP boot rom. After completing this message, the S facility no longer honors any messages until the monitor returns control. A k_null_reply is always returned for this message.

The S facility message structures are listed below:

```
typedef struct psa_msg {      /* A Message Template */
    SP_HEADER   header;
    vme_t       vme_addr;
    u_long      data_length;
    u_long      sram_addr;
    u_char      msg_body[K_MSG_SIZE - 32];
    void        (*rtnadr) ();   /* return address of
                                   a ready message */
    struct psa_msg *rblink;   /* points to a work area
                                 or msg link */
    u_long      start_time;
    } SP_MSG;
typedef struct {
    char        vme_bus_request_level;
    char        access_mode;
    char        number_of_disks;
    char        number_of_banks;
    short       firmware_revision;
    short       hardware_revision;
    int         total_secto[3];
    int         stripe_size[3];
    int         online_drive_bit_map[3]
    } config_data;
typedef struct {
    SP_HEADER   header;           /* byte 0-7 */
    config_data *vme_ptr;         /* byte 8-11 */
    long        data_length;      /* byte 12-15 sizeof
                                     config_data */
    } SEND_CONFIG_MSG;
typedef struct {
    SP_HEADER   header;           /* byte 0-7 */
    config_data *vme_pointer;
    long        data_length;
} RECEIVE_CONFIG_MSG;
typedef struct {
    SP_HEADER   header;                 /* byte 0-7 */
    char        scsi_id;                /* byte 8 */
    char        disk_number;            /* byte 9 */
    short       reserved;               /* byte 10-11 */
    short       sector_count;           /* byte 12-13 */
    short       emsc_sector_count;      /* byte 14-15 */
    long        sector_address;         /* byte 16-19 */
    u_long      vme_address;            /* byte 20-23 */
} SP_RDWR_MSG;
typedef struct {
    SP_HEADER   header;                 /* byte 0-7 */
    char        scsi_id;                /* byte 8 */
    char        disk_number;            /* byte 9 */
    short       reserved;               /* byte 10-11 */
    short       sector_count;           /* byte 12-13 */
    short       cache_page_size;        /* byte 14-15 */
    long        sector_address;         /* byte 16-19 */
    u_long      vme_address[10];        /* byte 20-23 */
} SP_RDWR_MSG;
typedef struct {
    SP_HEADER   header;                 /* byte 0-7 */
    char        scsi_id;                /* byte 8 */
    char        scsi_port;              /* byte 9 */
    char        scsi_lun_address;       /* byte 10 */
    char        command_length;         /* byte 11 */
    u_long      data length;            /* byte 12-15 */
    u_long      data_buffer_address;    /* byte 16-19 */
    char        command_bytes[20];      /* byte 20-39 */
    u_long      sense_length;           /* byte 40-43 */
    u_long      sense_addr;             /* byte 44-47 */
} SP_IOCTL_MSG;
```

IV. Start-up Opcrations

A. IFC Initialization

The chart below summarizes the system operations that occur during system boot.

TABLE 12

Summary of System Initialization

```
Phase 1: All peer-level processors

{
boot to "boot-level" ready state;
}
Phase 2: The host boot level facility

{
boot Unix image through boot-level S facility;
execute Unix image;
start SC_NAME_SERVER process;
}
Phase 3: The host facility

{
for each boot-level facility {
    probe for existence;
    initialize FIFO for receiving;
    }
for each ( SP NC FC ) {
    read boot image and parameters from boot-
        level S facility;
    download boot image and boot parameters
        (including the PID of the SC_NAME_SERVER
        process) to the shared memory program
        store of the peer-level processor;
    start controller;
    }
}
Phase 4: Each peer-level processor

{
begin executing facility image
    initialize controller
        send SC_REG_FIFO to SC_NAME_SERVER;
        send SC_GET_SYS_CONF to SC_NAME_SERVER;
        send SC_INIT_CMPL to SC_NAME_SERVER;
        }
    start manager processes {
        send SC_REG_NAMEs to SC_NAME_SERVER;
        send SC_RESOLVE_NAMEs to SC_NAME_SERVER;
        send SC_RESOLVE_FIFOs to SC_NAME_SERVER;
        }
    }
```

The SP peer-level processors boot from onboard EPROMs. The SP boot program, in addition to providing for power-on diagnostics and initialization to a ready state, includes a complete S facility. Thus, the SP peer-level processor is able to perform SCSI disk and tape operations upon entering its ready state. In their ready states, the NC, FC, SP and H processors can be downloaded with a complete instantiation of their respective types of facilities. The downloaded program is loaded into local shared memory; for the S facility, for example, the program is loaded into its local 256K static ram. The ram download, particularly to static ram, allows both faster facility execution and use of the latest release of the facility software.

After powering up or resetting the SP processor, the host facility, executing its boot program, waits for the SP boot program to post ready by indicating a ready state value in an SP status register.

Once the S boot program has posted ready, a Sector Read message from the host boot program can be used to retrieve any disk block to any VME memory location. Generally, the read request is to load the host facility from disk block **0**, the boot block. In preparing a read_sector message for the S facility after power up, the local host boot program specifies the following (in addition to normal read_sector message contents):

sender pid=0xffffffff dest_pid=0x00000001

By specifying the above, the local host boot program signals the S facility to bypass normal IFC reply protocols and to, in turn, signal a reply complete by directly by changing the 0xffffffff message value in the original message image to any other value, such as the value of the message descriptor. That is, after building a read sector message, the host boot program writes a message descriptor to the S facility. The host boot program can then poll this sender_pid word to determine when the message is completed. Messages to the S facility are sent in this manner until the full host facility boot is complete.

Once the local host boot program has loaded the host facility and begun executing its initialization, the host facility generally switches over to normal IPC communication with the S facility. To do this, local host facility sends an IFC Initialization message to the S facility. After receiving this message, the S facility expects a shared memory block, as specified by the message, to contain the following information:

- Byte **00–03**—Bootlock, provides synchronization with the local host facility
- Byte **04–05**—S facility board slot id,
- Byte **06–07**—Reserved,
- Byte **08–09**—This board's IFC virtual slot ID
- Byte **10–11**—System controller process number,
- Byte **12–27**—System controller fifo descriptor
- Byte **00–01**—System controller fifo type,
  - Byte **02–03**—System controller slot id
  - Byte **04–07**—Fifo address
  - Byte **08–09**—Soft fifo index,
  - Byte **10–11**—Soft fifo index mask,
  - Byte **12–13**—Interrupt request level,
  - Byte **14–15**—Interrupt vector address,
- Byte **28–31**—Address of this common memory, and
- Byte **32–35**—Size of this common memory.
- Byte **36–39**—Hardware fifo address of the S facility

The first thing the S facility does is check the bootlock variable. When it is set to a "BOOTMASTER" value, it means the local host facility is up and ready to receive message from the S facility. Otherwise, the S facility waits for the local host facility to complete its own initialization and set the bootlock word. As soon as the bootlock word is changed, the S facility proceeds to perform IFC initialization. The following IFC messages are sent to the local host facility:

1. Register FIFO
2. Get System Configuration
3. Initialization Complete
4. Register Name
5. Resolve FIFO

The second message allows the S facility to know who is in what VME slots within the system. The S facility will only register one name, "SPn" (n is either 0 or 1), with a processor ID of 1. Hence all messages directed to the S facility specify PID=SP_SLOT<<16+0x0001. Basically, a processor ID (PID) is a 4-byte word, in which the higher order two bytes contain the processor's VME slot ID. The lower order two bytes identify a process within a processor.

The register FIFO message formally informs the local host facility about the S facility's fifo address. The get system configuration message retrieves a table describing all available processors from the local host facility. After completing initialization, using the Initialization Complete message, the S facility advertises its services by issuing the

Register Name message, which informs the host facility that the S facility service process is up and running. When another facility sends a message to the S facility for the first time, the S facility uses a Resolve FIFO message, directed to the host facility, to obtain the fifo address needed for a reply.

Thus, a multiple facility operating system architecture that provides for the control of an efficient, expandable multi-processor system particularly suited to servicing large volumes of network file system requests has been described.

Clearly, many modifications in variations of the present invention are possible in light of the above teachings. Therefore, it is to be understood that within the scope of the appended claims, the principles of the present invention may be realized in embodiments other than as specifically described herein.

We claim:

1. A computer system employing a multiple facility operating system architecture, said computer system comprising:

a) a plurality of processor units provided to co-operatively execute a predetermined set of operating system peer-level facilities, wherein each said processor units is associated with a respective different one of said operating system peer-level facilities and not another of said operating system peer level facilities, and wherein each of said operating system peer level facilities constitutes a respective separately executed software entity which includes a respective distinct set of peer-level facility related functions, each said processor unit including:

i) a processor capable of executing a control program; and

ii) a memory store capable of storing said control program, said processor being coupled to said memory store to obtain access to said control program,

said memory store providing for the storage of a first control program portion that includes a one of said respective distinct sets of operating system peer-level facility related functions and that corresponds to a one of said predetermined operating system peer-level facilities, and a second control program portion that provides for the implementation of a multi-tasking interface function, said multi-tasking interface function being responsive to control messages for selecting for execution a one of said peer-level facility related functions of said one of said predetermined operating system peer-level facilities and responsive to said one of said predetermined operating system peer-level facilities for providing control messages to request or in response to the performance of said predetermined peer-level facility related functions of another operating system peer-level facility; and

b) a communications bus that provides for the interconnection of said plurality of processor units, said communications bus transferring said control messages between the multi-tasking interface functions of said predetermined set of operating system peer-level facilities.

2. The computer system of claim **1** wherein a first one of said predetermined set of operating system peer-level facilities includes a network communications facility and a second one includes a filesystem facility.

3. The computer system of claim **2** wherein said network communications facility is coupled to a network to permit the receipt of network requests, said network communications facility providing for the identification of a predetermined filesystem type network request, said multi-tasking interface function of said network communications facility being responsive to said predetermined filesystem type network request to provide a predetermined control message to said filesystem facility to request the performance of a predetermined filesystem function.

4. The computer system of claim **3** further comprising a data store that provides for the storage of data, said predetermined filesystem type network request directing said network communications facility to transfer predetermined data with respect to said network, said data store being coupled to said network communications facility for storing said predetermined data.

5. The computer system of claim **3** or **4** wherein said predetermined set of peer-level facilities further includes a storage facility and wherein said filesystem facility provides for the performance of said predetermined filesystem function, said multi-tasking interface function of said filesystem facility being responsive to said filesystem facility to provide control messages to said storage facility to request the performance of a predetermined storage access function.

6. The computer system of claim **5** wherein said predetermined storage access function directs said storage facility to transfer said predetermined data, said data store being coupled to said storage facility for storing said predetermined data.

7. A computer system implementing a co-operative facility based operating system architecture, said computer system comprising:

a) a plurality of processors, each being coupled to a respective control program store and a respective data store, said plurality of processors being interconnected by a communications bus; and

b) a multiple facility operating system having a kernel and providing for the message based co-operative operation of said plurality of processors, said multiple facility operating system providing for the operating system internal execution of a plurality of operating system peer-level facilities by execution of each of said peer-level facilities by a respective different one of said plurality of processors, each of said peer-level facilities constituting a respective software entity executed separately from said kernel, wherein each of said plurality of facilities implements a multi-tasking interface coupleable between said communications bus and a respective and unique peer-level control function set to permit message transfer between each of said plurality of facilities.

8. The computer system of claim **7** wherein said plurality of facilities includes a network facility and a filesystem utility, wherein said network facility includes a communications network peer-level control function coupled between a first multi-tasking interface and a network interface and said filesystem facility includes a data storage peer-level control function coupled between a second multi-tasking interface and a filesystem.

9. The computer system of claim **8** wherein said network facility is coupled through said network interface to a communications network, wherein said network facility is responsive to a predetermined network filesystem message received via said network interface to provide a predetermined filesystem message, and wherein said filesystem facility is responsive to said predetermined filesystem message to transfer data with respect to said filesystem.

10. The computer system of claim **9** further comprising a common data store, said network facility providing for the transfer of data between said network interface and said data store, said filesystem facility providing for the transfer of

data between said data store and said filesystem, said communications network peer-level control function directing a message to said filesystem peer-level control function identifying a predetermined location of data in said data store with respect to said predetermined filesystem message.

11. A computer system employing a multiple facility operating system to provide for co-operative operation of a plurality of processors,

wherein said operating system includes a kernel and a plurality of additional component facilities executed separately from said kernel, each of said component facilities including a facility sub-component, that defines the execution operation of a one of said component facilities, coupled to a multi-tasking interface sub-component,

wherein said computer system comprises:

a) a plurality of processors executing said operating system, each of said processors including local memory for the storage and execution of a respective component facility;

b) a data memory accessible by each of said processors for the storage and retrieval of data blocks exchangeable between said processors; and

c) a communications bus coupling said processors and said data memory to permit the exchange of control messages between said processors and data through said data memory,

and wherein said processors each implement a respective different local sub-set of fewer than all of said component facilities that depends through the exchange of control messages on the execution of another sub-set of said componentized facilities by another of said processors to co-operatively implement said operating system.

12. The computer system of claim **11** wherein control messages communicate any of a facility sub-component function request, a facility sub-component function response, and a facility sub-component identifier of a memory space within said data memory to use in connection with said sub-component function request.

13. The computer system of claim **12** wherein said plurality of component facilities includes a network facility and a filesystem facility, wherein a network facility sub-component is executed by a first processor to process network requests and data transfers and a filesystem facility sub-component is executed by a second processor to process filesystem requests and data transfers derivative of said network requests and data transfers.

14. The computer system of claim **1**, wherein one of the processor units in said plurality of processor units is provided further to execute a further operating system peer-level facility not in said predetermined set of operating system peer-level facilities.

15. The computer system of claim **7**, wherein said multiple facility operating system provides further for the operating system internal execution of a further operating system peer-level facility not in said plurality of operating system peer-level facilities, by execution of said further peer level facility by one of the processors in said plurality of processors.

16. The computer system of claim **7**, wherein said kernel is a Unix kernel.

17. The computer system of claim **11**, wherein said kernel is a Unix kernel.

* * * * *

X. RELATED PROCEEDINGS APPENDIX

None.